Faculty of Informatics
Masaryk University

MINIMAL SETS OVER A MONOTONE PREDICATE: ENUMERATION AND COUNTING

Jaroslav Bendík

PHD THESIS

Brno, 2020

Advisor: Ivana Černá, Masaryk University, Brno

Reviewer #1: Nikolaj Bjørner, Microsoft Research, Redmond
Reviewer #2: João Marques-Silva, IRIT/CNRS, Toulouse

The thesis was successfully defended on the 22nd of March, 2021. This is a revised version that reflects comments provided by the reviewers. The complete archive of the thesis, including the reviews, is available at `https://is.muni.cz/th/y4v8m/?lang=en`.

Acknowledgements

There are many people who supported me during my studies and who deserve my gratitude. Big thanks go to my advisor, Ivana, who guided my steps on the academic ground already since my bachelor studies. Not only she provided me with all the necessary administrative, funding, and other support I needed, but she also taught me how to be a good researcher. Moreover, she was always willing to discuss with me both academic and personal life matters, and thus I can call her both my advisor and my friend.

I am equally grateful to Jiřík, who allowed me to participate in two European research projects where I had the opportunity to collaborate with research teams from both academy and industry. Even more than the research and academic-related stuff, I enjoyed friendly conversations on numerous topics we had on a daily basis. Also, I am thankful for Jiřík's endeavor to teach me how to play squash despite my constant injuries; I hope one day we will get the chance to play again.

I also want to thank Strejda for numerous friendly discussions and for both life and academic pieces of advice he gave me. To Nikola, Mike, and Kuldeep, I am grateful for fruitful cooperation on several research projects and for sharing with me their academic and other experiences. To all members of the ParaDiSe laboratory, both present and former, I am grateful for creating a friendly and inspiring working atmosphere.

Finally, it would be hardly possible to finish my studies without being supported by my family. In particular, I am grateful to my wife who was always willing to hear me out when I needed it. She was never angry with me when I was chasing a deadline and devoted all my days and nights to work instead of spending my time with her. Moreover, she always cheered me up when I was not in the mood. I am also thankful to my son since he was the one who finally made me change my daily schedule and strictly separate the working and non-working hours. Last but not least, I am grateful to my parents since they always supported me, offered me a shelter I can return to, and encouraged me in everything reasonable I wanted to do.

Abstract

In many areas of computer science, we are given a reference set C and a monotone predicate $\mathbf{P} : \mathcal{P}(C) \rightarrow \mathit{Bool}$. The monotonicity means that if $\mathbf{P}(N) = \mathit{True}$ then $\mathbf{P}(M) = \mathit{True}$ for every $N \subseteq M \subseteq C$. The goal is to identify the **M**inimal **S**ubsets of C satisfying the **M**onotone **P**redicate (MSMPs). For instance, assume that C is an unsatisfiable set of Boolean clauses and $\mathbf{P}(N) = \mathit{True}$ iff $N \subseteq C$ is unsatisfiable. In this case, the MSMPs are the *minimal unsatisfiable subsets* (MUSes) of C which are often used during an analysis of unsatisfiable Boolean formulas. The contribution of the thesis is the following.

In the first part, we propose two *domain agnostic* algorithms for MSMP enumeration, i.e. algorithms that can be applied for any particular instance of MSMPs. A natural role of domain agnostic algorithms is to serve as ready-to-use solutions for any new instance of MSMPs that might arise.

In the second part, we focus on the particular instance of MSMPs where the input set C is a set of Boolean clauses and the predicate $\mathbf{P}$ is the Boolean unsatisfiability. We propose a novel algorithm for enumeration of MUSes of C. Moreover, we propose a novel algorithm for enumeration of maximal satisfiable subsets (MSSes) of C.

In the third part, we examine the problem of counting the MUSes and MSSes of C. The existing approaches for MUS/MSS counting rely on complete MUS/MSS enumeration, however, for instances with many MUSes/MSSes, the complete enumeration is often practically intractable. We present the very first MUS and MSS counting algorithms that determine the MUS and MSS count, respectively, without explicit MUS and MSS enumeration.

In the fourth part, we study another instance of MSMPs called *minimal inductive validity cores* (MIVCs). The input is a transition system and a safety property satisfied by the system, i.e., a set of unreachable unsafe states. The transition relation of the system is encoded as a conjunction of a set $\{T_1, \ldots, T_n\}$ of transition step predicates. An MIVC of the system is a minimal subset of the predicates that need to be left in the system to satisfy the safety property. We propose a novel MIVC enumeration algorithm.

In the last part, we are given as an input a timed automaton (TA), a set C of clock constraints that restrict the transitions of the TA, and a set L of unreachable locations of the TA. Our goal is to identify a *minimal sufficient reduction* of the TA, i.e,. a minimal subset of the clock constraints C that needs to be relaxed to make L reachable.

Contents

List of Definitions

1 Introduction

In various areas of computer science, such as requirements analysis, model checking, or formal equivalence checking, the following situation arises. We are given a set of constraints with a goal to decide whether the set of constraints is satisfiable, i.e., whether all the constraints can hold simultaneously. In case the given set is shown to be unsatisfiable, we might be interested in analyzing the unsatisfiability. Identification of minimal unsatisfiable subsets (MUSes) of the unsatisfiable set of constraints is a kind of such analysis.

In the requirements analysis, constraints represent the requirements on the system that is being developed, and checking for satisfiability means checking whether all the requirements can be implemented at once. If the set of requirements is unsatisfiable, an identification of MUSes helps to identify and fix conflicts among requirements [Barnat et al., 2016]. In some model checking approaches, e.g., the counterexample-guided abstraction refinement (CEGAR) [Clarke et al., 2000], an unsatisfiable set of constraints may arise as a result of the abstraction's overapproximation. In such a case, the extraction of MUSes leads to a better refinement of the overapproximation [Andraus et al., 2008].

Besides the problems defined in terms of *constraints* and *satisfiability*, there are many problems with the same underlying structure, however, using different terminology. For example, see the problems of finding *minimal correction subsets* [Bailey and Stuckey, 2005], *minimal independent supports* [Ivrii et al., 2016], *minimal inconsistent subsets* [Barnat et al., 2016], or *minimal strongly inconsistent programs* [Mencía and Marques-Silva, 2020]. In general, in all these problems, we are given on input a set C of elements and a *monotone* predicate $\mathbf{P} : \mathcal{P}(C) \rightarrow Bool$ that classifies every subset of C. The monotonicity means that if $\mathbf{P}(N) = True$ then $\mathbf{P}(N') = True$ for every $N \subseteq N' \subseteq C$. The goal is to identify the **M**inimal **S**ubsets of C satisfying the **M**onotone **P**redicate (MSMPs) [Marques-Silva et al., 2013b, 2017]. For instance, in the cases of MUSes, the set C contains the constraints (e.g., Boolean formulas) and the predicate $\mathbf{P}$ holds for unsatisfiable subsets of C.

In this thesis, we deal with several functional problems that can be formulated in the terms of minimal sets over a monotone predicate. In particular, our research focuses on the following five areas.

Domain Agnostic MUS Enumeration In the first part of the thesis, we focus on the subclass of MSMPs where we are given an *unsatisfiable set* C of *constraints* with the goal to enumerate the minimal unsatisfiable subsets (MUSes) of C. Perhaps most of the existing applications of MUSes work either with Boolean (SAT) or SMT constraints; these types of constraints arise, for example, in the CEGAR workflow or in the area of constraint processing. In software requirements, the most common constraints are those expressed in some temporal logic like Linear Temporal Logic (LTL) [Pnueli, 1977], Metric Temporal Logic (MTL) [Koymans, 1990], or Computation Tree Logic (CTL) [Clarke and Emerson, 1981].

Since the list of constraint domains in which MUS enumeration finds an application is quite long and new applications still arise, there is a desire for *domain agnostic* MUS enumeration algorithms. Such algorithms can be used in an arbitrary constraint domain, and thus serve as ready-to-use solutions for any constraint domain where MUSes might eventually find an application. A domain agnostic algorithm cannot directly exploit domain specific properties of particular constraint domains. However, there are some properties that are common for almost all constraint domains where MUSes find applications. For example, the evaluation of the monotone predicate, i.e., performing the satisfiability checks, is usually very expensive (often NP-complete or harder). Thus, domain agnostic MUS enumeration algorithms should tend to minimize the number of performed satisfiability checks.

In this thesis, we present two domain agnostic MUS enumeration algorithms that tend to reduce the number of performed checks by directly exploiting general properties of MUSes, e.g., the monotonicity of the satisfiability function, and indirectly via domain specific black-box subroutines. We show, via an experimental evaluation in multiple constraint domains, that our algorithms are competitive or even superior to other contemporary approaches.

MUS and MSS Enumeration in the Boolean CNF Domain In the second part of the thesis, we focus on the particular instance of MUSes where we are given on input an unsatisfiable set C of Boolean clauses, or equivalently, an unsatisfiable Boolean formula in the conjunctive normal form (CNF formula). The goal is to enumerate all MUSes of C. The application areas of Boolean CNF MUS identification include for example ontologies debugging [Arif et al., 2016], diagnosis [Mu, 2019], spreadsheet debugging [Jannach and Schmitz, 2016], constrained counting and sampling [Ivrii et al., 2016], formal equivalence checking [Cohen et al., 2010], and the like.

Many of contemporary MUS enumeration algorithms implement a *seed-shrink* scheme. During their computation, the algorithms gradually *explore* individual subsets of the input set C of clauses. Explored subsets are those, whose satisfiability has been already determined by the algorithm, and unexplored are the others. To find each single MUS, a seed-shrink algorithm first identifies a *seed*, i.e., an unexplored unsatisfiable subset of C. Subsequently, the seed is *shrunk* to an MUS using a single MUS extraction procedure. In general, the algorithms identify a seed by repeatedly picking and checking unexplored subsets for satisfiability (via a SAT solver), until they find an unsatisfiable one. The difference between the algorithms is in *which* subsets they check for satisfiability, and also *how* exactly they perform the single MUS extraction.

In this thesis, we propose a novel MUS enumeration algorithm that is based on the seed-shrink scheme. In a significant departure from the contemporary seed-shrink approaches, our algorithm identifies a vast majority of seeds via a cheap (polynomial) deduction technique. Consequently, our algorithm is much more frugal in the number of performed satisfiability checks, which allows it to outperform other contemporary solutions.

Moreover, we propose a novel algorithm for enumeration of *maximal satisfiable subsets (MSSes)* of a given unsatisfiable set of Boolean clauses. MSSes are natural counter-part of MUSes, and they also find applications during various tasks such as model-based diagnosis [Ben-Eliyahu and Dechter, 1993], axiom pinpointing [Arif et al., 2015a], or MaxSAT solving [Marques-Silva et al., 2013a]. Similarly to our MUS enumeration algorithm, our MSS enumeration algorithm employs a novel deduction technique that allows it to significantly reduce the number of performed satisfiability checks during the MSS enumeration.

MUS and MSS Counting in the Boolean CNF Domain Yet another problem we focus on is counting the number of MUSes and/or MSSes of a given set C of Boolean clauses. Current applications of MUS and MSS counting include mainly various inconsistency metrics for general propositional knowledge bases [Thimm, 2018, Hunter and Konieczny, 2008, Mu, 2019]. Compared to the problems of single MUS extraction and MUS enumeration that have been extensively studied in the past two decades, the research on MUS and MSS counting is still in its nascent stage. The contemporary approach is to simply enumerate all MUSes (MSSes). However, since the number of MUSes (MSSes) can be exponential in the number of clauses in C, the complete MUS (MSS) enumeration is often practically intractable within a reasonable time limit. In this context, one wonders: *whether it is possible to design a scalable MUS (MSS) counter that does not rely on explicit MUS (MSS) enumeration*.

The thesis provides an affirmative answer to the above question. In particular, we present a probabilistic MUS counter that takes as an input a CNF formula C, a tolerance parameter ε, a confidence parameter δ, and it returns an estimate guaranteed to be within $(1+\varepsilon)$-multiplicative factor of the exact count with confidence at least $1-\delta$. Crucially, if C contains n clauses, our algorithm explicitly identifies only $\mathcal{O}(\log n \cdot \log(1/\delta) \cdot (\varepsilon)^{-2})$ many MUSes even though the number of MUSes can be exponential in n. As for MSSes, we introduce a reduction of the exact MSS counting problem to the problem of projected propositional model counting. The reduction allows us to exploit recent advances in the design of efficient component caching-based projected model counting techniques. We experimentally show that both our MUS and MSS counting approaches scale much better than contemporary MUS and MSS enumeration algorithms.

Proof Explanation in Symbolic Model Checking An interesting instance of minimal sets over a monotone predicate emerges in the area of symbolic model checking. Automated model checking techniques such as IC3/PDR [Eén et al., 2011], k-induction [Sheeran et al., 2000], and k-liveness [Claessen and Sörensson, 2012] can be used to determine whether safety properties hold of finite or even infinite-state systems. Moreover, if a safety property is violated in a system, these techniques provide a counter-example, i.e. a trace in the system, demonstrating a situation in which the property fails to hold. However, if a property is proved to hold, most model checking tools do not provide any *human-readable* explanation of why the property holds. Consequently, developers might have an unwarranted level of confidence in the behavior of the system, and issues such as vacuity [Kupferman and Vardi, 2003] or incorrect environmental assumptions [Whalen et al., 2007] can lead to failures of "proved" systems.

To explain why a safety property holds in a system in a formal, yet human-readable way, Ghassabani et al. proposed the concept of *Minimal Inductive Validity Cores (MIVCs)* [Ghassabani et al., 2016]. They assume that the transition relation of the system is encoded as a conjunction of a set $T = \{T_1, \ldots, T_n\}$ of transition step predicates. Intuitively, the addition of each individual transition step predicate to the system makes some states to be unreachable. An MIVC of the system is a minimal subset T' of T such that the system induced by the transition steps predicates T' still satisfies the safety property. Enumeration of MIVCs can be used, e.g., during coverage analysis, optimizing logic synthesis, impact analysis, or robustness analysis [Ghassabani et al., 2017b]. In the thesis, we propose a novel MIVC enumeration algorithm and experimentally compare it with other contemporary approaches.

Relaxing Timed Automata for Reachability In the last part of the thesis, we introduce a novel instance of minimal sets over a monotone predicate, called *minimal sufficient reductions (MSRs)*, that find an application in the area of timed automata. A timed automaton (TA) is a finite automaton extended with a set of real-time variables, called clocks, which capture the time and control/restrict the behavior of the automaton. Examples of TA models of time-critical systems include, e.g., scheduling of real-time systems [Fehnker, 1999, David et al., 2009, Guan et al., 2007], medical devices [Kwiatkowska et al., 2015, Jiang et al., 2014], and rail-road crossing systems [Wang, 2004].

Contemporary model-checking tools, such as UPPAAL [Behrmann et al., 2006] or IMITATOR [André et al., 2012], allow for verifying whether a given TA satisfies a system specification. Unfortunately, during the system design phase, the system information is often incomplete. A designer is able to build a TA with a correct structure, i.e., correctly capturing locations and transitions of the system, however, the exact clock constraints that enable/trigger the transitions are uncertain. Consequently, the produced TA often does not meet the specification and needs to be refined.

In case of violation of a universal property, such as safety or unavoidability, that needs to hold on each trace of the system, a model checker provides a counter-example, i.e., a trace on which the property is not satisfied. Subsequently, the counter-example can be used to refine the model. However, in case of violation of an existential property, e.g., reachability, that needs to hold on a trace of the system, the model checker is not able to provide any information that would help the designer to correct the TA.

In the thesis, we propose a novel technique for relaxing timed automata for reachability properties. In particular, given a timed automaton $\mathcal{A}$, we first identify a *minimal sufficient reduction (MSR)* of $\mathcal{A}$, i.e., a minimal subset S of clock constraints of $\mathcal{A}$ that need to be relaxed to satisfy the reachability property. In the second step, we employ either linear programming or a parameter synthesis tool to actually find a relaxation of S that leads to a satisfaction of the reachability property.

1.1 *Structure of the Thesis*

Chapter 2 defines the basic notation and concepts used throughout the thesis. Especially, it formally defines the general concept of minimal sets over a monotone predicate (MSMPs) and states several well-known observations and techniques related to MSMP identification. The rest of the thesis is divided into the five parts as discussed above.

In the first part, we focus on the domain agnostic MUS enumeration. In particular, Chapter 3 provides a detailed description of several applications of MUSes taken from various constraint domains. The goal of the chapter is to motivate the development of domain agnostic MUS enumeration algorithms, and it also demonstrates how different the individual constraint domains can be. In Chapter 4, we present our two novel domain agnostic MUS enumeration algorithms, called TOME [Bendík et al., 2016b] and ReMUS [Bendík et al., 2018b], discuss other existing solutions, and provide an exhaustive experimental evaluation.

In the second part, we focus specifically on the case where we are given on input an unsatisfiable Boolean formula in CNF, i.e., an unsatisfiable set of Boolean clauses. Chapter 5 presents our Boolean CNF MUS enumeration algorithm called UNIMUS [Bendík and Černá, 2020c], and Chapter 6 introduces our Boolean CNF MSS enumeration algorithm called RIME [Bendík and Černá, 2020b]. Both chapters discuss also other contemporary MUS and MSS enumeration techniques and provide an experimental evaluation.

The third part deals with the problems of Boolean CNF MUS and MSS counting, and it is divided into Chapters 7 and 8. The former chapter presents our approximate MUS counting algorithm called AMUSIC [Bendík and Meel, 2020]. The latter chapter introduces our reduction [Bendík and Meel, 2021] of the exact MSS counting problem to the problem of projected propositional model counting.

In the fourth part, i.e., Chapter 9, we present our algorithm, called GROW-SHRINK [Bendík et al., 2018c], for enumerating minimal inductive validity cores (MIVCs). We also provide an overview of other existing MIVC enumeration techniques and show results of our experimental evaluation.

The final part of the thesis is formed by Chapter 10 and it introduces the concept of *minimal sufficient reductions (MSR)* [Bendík et al., 2021]. We present an algorithm for computing MSRs and we show how MSRs can be used during the relaxation of timed automata for reachability properties. We illustrate the usefulness of our approach on a case study and we compare it with another available timed automata relaxation technique.

1.2 *Author's Publications and His Contribution*

1.2.1 *Core of the Thesis*

Chapters of the dissertation thesis are based on conference publications co-authored by me, the author of the thesis. Below, I list the publications respecting the order of corresponding chapters. I am the leading and also the corresponding author of all the publications. Moreover, for each publication, I state my contribution in percentage. In particular, the contribution to each paper is divided into three categories: 1) origin of ideas behind the publication and participation in discussions with co-authors (*ideas and discussions*), 2) implementation and evaluation of the proposed methods (*implementation*), and 3) work on the text of the paper (*text*). In each of these categories, an author's contribution ranges between 0% and 100%. The *overall contribution* of an author is computed as a weighted sum of the three categories using weights 0.5, 0.25, and 0.25, respectively.[1]

[1] For instance, if an author contributed to *ideas and discussions*, *implementation*, and *text* with 70%, 60% and 50%, respectively, then his overall contribution is $70 \cdot 0.5 + 60 \cdot 0.25 + 50 \cdot 0.25 = 62.5\%$.

ISSTA 2017 (doctoral symposium) Jaroslav Bendík
"Consistency Checking in Requirements Analysis" [Bendík, 2017]
My contribution: ideas and discussions 100%, implementation 100%, text 100%, overall contribution **100.0**%.

FSTTCS 2016 Jaroslav Bendík, Nikola Beneš, Ivana Černá, and Jiří Barnat
"Tunable Online MUS/MSS Enumeration" [Bendík et al., 2016b]
My contribution: ideas and discussions 30.0%, implementation 100%, text 50.0%, overall contribution **52.5**%.

ATVA 2018 Jaroslav Bendík, Ivana Černá, and Nikola Beneš
"Recursive Online Enumeration of All Minimal Unsatisfiable Subsets" [Bendík et al., 2018b]
My contribution: ideas and discussions 50.0%, implementation 100%, text 50.0%, overall contribution **62.5**%.

LPAR 2018 Jaroslav Bendík and Ivana Černá
"Evaluation of Domain Agnostic Approaches for Enumeration of Minimal Unsatisfiable Subsets" [Bendík and Černá, 2018]
My contribution: ideas and discussions 60.0%, implementation 100%, text 80.0%, overall contribution **75.0**%.

TACAS 2020 Jaroslav Bendík and Ivana Černá
"MUST: Minimal Unsatisfiable Subsets Enumeration Tool" [Bendík and Černá, 2020a]
My contribution: ideas and discussions 75.0%, implementation 100%, text 95.0%, overall contribution **86.2**%.

CP 2020 Jaroslav Bendík and Ivana Černá
"Replication-Guided Enumeration of Minimal Unsatisfiable Subsets" [Bendík and Černá, 2020c]
My contribution: ideas and discussions 80.0%, implementation 100%, text 80.0%, overall contribution **85.0**%.

LPAR 2020 Jaroslav Bendík and Ivana Černá
"Rotation Based MSS/MCS Enumeration"
[Bendík and Černá, 2020b]
My contribution: ideas and discussions 80.0%, implementation 100%, text 80.0%, overall contribution **85.0**%.

CAV 2020 Jaroslav Bendík and Kuldeep S. Meel
"Approximate Counting of Minimal Unsatisfiable Subsets" [Bendík and Meel, 2020]
My contribution: ideas and discussions 50.0%, implementation 100%, text 50.0%, overall contribution **62.5**%.

AAAI 2021 Jaroslav Bendík and Kuldeep S. Meel
"Counting Maximal Satisfiable Subsets" [Bendík and Meel, 2021][2]
My contribution: ideas and discussions 75.0%, implementation 100%, text 75.0%, overall contribution **81.2**%.

[2] The paper was accepted to AAAI'21 after the official submission of the thesis and before the defence. In the original thesis submission, the paper was marked as "under a review" without specifying the conference.

SEFM 2018 Jaroslav Bendík, Elaheh Ghassabani, Michael W. Whalen, and Ivana Černá
"Online Enumeration of All Minimal Inductive Validity Cores" [Bendík et al., 2018c]
My contribution: ideas and discussions 50.0%, implementation 50.0%, text 50.0%, overall contribution **50.0**%.

TACAS 2021 Jaroslav Bendík, Ahmet Sencan, Ebru Aydin Gol, and Ivana Černá
"Timed Automata Relaxation for Reachability" [Bendík et al., 2021][3]
My contribution: ideas and discussions 40.0%, implementation 30%, text 50.0%, overall contribution **40.0**%.

[3] The paper was accepted to TACAS'21 after the official submission of the thesis and before the defence. In the original thesis submission, the paper was marked as "under a review" without specifying the conference.

A large portion of the text of the thesis and the structure of several chapters is adopted from the corresponding conference papers. However, some of the material was completely rewritten and some parts were substantially extended. In particular,

- Chapters 1, 2 and 3 are new.
- The description of our domain agnostic MUS enumeration algorithms TOME and ReMUS and our domain agnostic MUS enumeration tool MUST (Chapter 4) was completely rewritten to reflect various improvements we made since the publication of the original papers.
- The presentation of our Boolean CNF MUS enumeration algorithm in Chapter 5 is mostly adopted from the corresponding conference paper [Bendík and Černá, 2020c].
- The description of our Boolean CNF MSS/MCS enumeration algorithm RIME in Chapter 6 is mostly adopted from the corresponding paper [Bendík and Černá, 2020b]. However, we provide here a new experimental evaluation that captures low-level improvements of the implementation of the algorithm we made since the publication of the original paper. Moreover, we examine more criteria in the evaluation than in the original experimental evaluation.
- Chapter 7 that presents our MUS counting algorithm adopts a large portion of the text from [Bendík and Meel, 2020], however, it is extended by a description of subroutines of our algorithm that did not appear in the original paper [Bendík and Meel, 2020] (due to a page limit).
- Chapter 9 that describes our MIVC enumeration approach adopts a large portion of the text from the corresponding paper [Bendík et al., 2018c], however, it is extended with illustration of MIVCs in Lustre programs.
- Chapters 8 and 10 are mostly adopted from the corresponding papers.

Moreover, the notation, style of plots, figures and algorithms, etc., were unified in all chapters. Also, introductions and conclusions of the individual chapters and sections that discuss related work were substantially rewritten so that the thesis provides a coherent presentation.

Tools. All algorithms we present in this thesis were implemented in publicly available tools. I am either the only author or a co-author of all the tools (as stated in the contribution to the papers above). In particular, our domain agnostic MUS enumeration algorithms, TOME [Bendík et al., 2016b] and ReMUS [Bendík et al., 2018b], are implemented in our domain agnostic MUS enumeration tool MUST [Bendík and Černá, 2020a][4]. GROW-SHRINK [Bendík et al., 2018c], our algorithm for enumeration of Minimal Inductive Validity Cores, is available in a development branch[5] of the model checker JKind [Gacek et al., 2018]. Our Boolean CNF MUS and MSS enumeration algorithms UNIMUS [Bendík and Černá, 2020c] and RIME [Bendík and Černá, 2020b], respectively, are implemented

[4] https://github.com/jar-ben/mustool

[5] https://github.com/janetlj/jkind

in standalone tools[6,7]. Similarly, our Boolean CNF MUS and MSS counting algorithms AMUSIC [Bendík and Meel, 2020] and CountMSS, respectively, are provided as standalone tools[8,9]. Finally, our approach for relaxation of timed automata is implemented in a tool called Tamus[10].

[6] https://github.com/jar-ben/unimus
[7] https://github.com/jar-ben/rime
[8] https://github.com/jar-ben/amusic
[9] https://github.com/jar-ben/MSSCounting
[10] https://github.com/jar-ben/tamus

Inter-University Cooperation. Note that some of the research presented in this thesis was conducted with people from other universities I established cooperation with. In particular, the algorithm for MIVC enumeration [Bendík et al., 2018c] was developed mainly during my two-week visit of Mike Whalen and Elaheh Ghassabani at the University of Minnesota. The research on approximate MUS counting [Bendík and Meel, 2020] was conducted jointly with Kuldeep Meel during my four-week stay at the National University of Singapore. Follow-up cooperation with dr. Meel led to the development of the MSS counting technique presented in Chapter 8. Finally, the research on the relaxation of timed automata (Chapter 10) is joint work with Ebru Aydin Gol and Ahmet Sencan from the Middle East Technical University and it was initiated when dr. Gol visited my advisor at my university.

1.2.2 *Other Related Publications*

The author during his Ph.D. studies also examined possible generalizations of the problem of MSMP identification via a concept of an *ADAG*. An *ADAG* is a directed acyclic graph annotated with a Boolean validation function that classifies each vertex of the graph either as *valid* or as *invalid*. A vertex v of a given ADAG is a *minimal valid vertex* (MVV) if v is valid and every its direct predecessor is *invalid*. Symmetrically, one can define *maximal invalid vertices*. Given a monotone predicate P over a reference set C, the problem of finding minimal subsets of C satisfying the monotone predicate P can be straightforwardly mapped to the problem of finding MVVs in an ADAG. In particular, the graph is induced by the power-set of C and the validation function corresponds to the monotone predicate P. When modeling the problem of MSMP identification, the induced graph has a specific structure of a hypercube graph (the structure of a power-set). Moreover, the validation function is *monotone* w.r.t. the reachability relation since the predicate P is monotone w.r.t. the subset inclusion relation. The author examined the problem of MVV identification even in the more general cases where the assumption about the structure of the graph or the monotonicity assumption is relaxed. The results were published in the following two papers.

SEFM 2016 Jaroslav Bendík, Nikola Beneš, Jiří Barnat, and Ivana Černá
"Finding Boundary Elements in Ordered Sets with Application to Safety and Requirements Analysis" [Bendík et al., 2016a][11]
My contribution: ideas and discussions 30.0%, implementation 100%, text 50.0%, overall contribution **52.5**%.

[11] In this paper, we focused on finding MVVs in ADAGs that can have a structure of an arbitrary acyclic directed graph and the validation function is monotone w.r.t. the reachability relation.

ICSOFT 2018 Jaroslav Bendík, Nikola Beneš, and Ivana Černá
"Finding Regressions in Projects under Version Control Systems" [Bendík et al., 2018a][12]
My contribution: ideas and discussions 50.0%, implementation 100%, text 50.0%, overall contribution **62.5**%.

[12] In this paper, we examined the class of ADAGs where the underlying graph can be an arbitrary directed acyclic graph and, moreover, the validation function does not have to be monotone w.r.t. the reachability relation. Such a structure naturally arises for example in commit graphs of version control systems: the vertices correspond to individual commits and the validation function classifies the status of individual commits (e.g., if the commits pass performance tests).

Since the two papers focus on a generalization of the MSMP identification problem, the algorithms presented in the papers can be naturally used also for finding MSMPs. However, due to the generalization, the algorithms cannot fully exploit properties that are specific for MSMP identification and thus are rather inefficient for that task. Therefore, the thesis does not discuss the two papers in more detail.

2 Preliminaries

Given a set X, we use $\mathcal{P}(X)$ to denote the power-set of X, and $|X|$ to denote the cardinality of X. For a pair X, Y of sets, we write $X \subseteq Y$ to denote that X is a subset of Y, and $X \subsetneq Y$ to denote that X is a proper subset of Y. In writing, we use "iff" as a shorthand for the equivalence relation "if and only if". In proofs of such equivalence statements, we use $\Leftarrow$ and $\Rightarrow$ to split the two implication directions. In case of propositional implications, we use the symbols $\leftarrow$ and $\rightarrow$ (see Section 2.1).

2.1 Propositional Formulae

Standard propositional logic definitions are used throughout the thesis (e.g. [Kleine Büning and Lettmann, 1999]), some of which are introduced in this section. When needed, additional definitions are introduced in the particular chapters of the thesis.

An atom is a propositional variable. A literal is either a variable x or its negation $\neg x$. A propositional formula is defined inductively over a set of propositional variables and standard logical connectives, $\neg$, $\wedge$ and $\vee$, as follows:

1. Every atom is a formula.

2. If F is a formula, then the negation $(\neg F)$ of F is a formula.

3. If F and G are formulae, then the conjunction $(F \wedge G)$ is a formula.

4. If F and G are formulae, then the disjunction $(F \vee G)$ is a formula.

We also use the standard extensions of the inductive definition using the connectives of $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence). When clear from the context, we simply write *formula*, instead of the full term *propositional formula*. Moreover, we use the terms *propositional formula* and *Boolean formula* interchangeably. Parentheses are used in a formula only when needed for clarity of the presentation (standard binding priorities are used, i.e., $\neg$ binds more tightly than $\vee$ and $\wedge$, $\vee$ and $\wedge$ bind more tightly than $\rightarrow$, and $\rightarrow$ binds more tightly than $\leftrightarrow$).

Given a formula F, we use $\mathit{Vars}(F)$ to denote the set of variables of F. A truth assignment, also called valuation or variable assignment,

π to a set A of variables is a mapping $\pi : A \to \{True, False\}$. If $A \supseteq Vars(F)$, the value of F w.r.t. π, denoted F^{π}, is defined as follows:

1. if $F = x$ where x is a variable, then $F^{\pi} = \pi(x)$
2. if $F = \neg G$ then
$$F^{\pi} = \begin{cases} False & \text{if } G^{\pi} = True \\ True & \text{if } G^{\pi} = False \end{cases}$$
3. if $F = G \vee H$ then
$$F^{\pi} = \begin{cases} True & \text{if } G^{\pi} = True \text{ or } H^{\pi} = True \\ False & \text{otherwise} \end{cases}$$
4. if $F = G \wedge H$ then
$$F^{\pi} = \begin{cases} True & \text{if } G^{\pi} = True \text{ and } H^{\pi} = True \\ False & \text{otherwise} \end{cases}$$

If $F^{\pi} = True$, we write $\pi \models F$ and say that π satisfies F. We also call π a satisfying assignment of F or a model of F. In the other case, when $F^{\pi} = False$, we write $\pi \not\models F$ and say that π does not satisfy F. A formula F is satisfiable if it has a model; otherwise, it is unsatisfiable.

Given a formula F, a truth assignment π to $Vars(F)$ can be identified with the set $trues(\pi, F) = \{x \in Vars(F) \mid \pi(x) = True\}$ of variables that are assigned *True*. Through the lens of this set representation, we can define *minimal* and *maximal* models of a formula as follows.

Definition 2.1 (minimal model). *Given a formula F, a model π of F is* minimal *iff there there is no model π' of F such that $trues(\pi', F) \subsetneq trues(\pi, F)$.*

Definition 2.2 (maximal model). *Given a formula F, a model π of F is* maximal *iff there there is no model π' of F such that $trues(\pi, F) \subsetneq trues(\pi', F)$.*

A *clause* is a disjunction of literals $l_1 \vee l_2 \vee \cdots \vee l_n$. A formula F is in a conjunctive normal form (CNF) iff it is formed by a conjunction of clauses. We shortly call such formulae CNF formulae. A CNF formula can be identified with the set of its clauses and a clause can be identified with the set of its literals. We use the two notations (i.e., the set notation and the one using conjunctions and disjunctions) interchangeably; the currently used notation is always clear from the context. Note that a truth assignment satisfies a CNF formula iff it satisfies all clauses of the formula. Similarly, a truth assignment satisfies a clause iff it satisfies a literal of the clause.

We use the following well-known *monotonicity* properties of CNF formulae (see, e.g., [Kleine Büning and Lettmann, 1999]):

Observation 2.1. *Let F and G be CNF formulae such that $F \subseteq G$, and let π be a model of G. Then π is also a model of F. Consequently, if G is satisfiable then F is also satisfiable.*

Observation 2.2. *Let F and G be CNF formulae such that $F \subseteq G$. If F is unsatisfiable then G is also unsatisfiable.*

The above two observations are widely used in the thesis, however, since they are well-known, they will not be explicitly referred to for the sake of simplicity.

Finally, given two CNF formulae C and F with $F \subseteq C$, we exploit the capabilities of contemporary SAT solvers to provide either an *unsat core* of F when F is unsatisfiable, or a model of F and the corresponding *model extension* of F w.r.t. C when F is satisfiable:

Definition 2.3 (unsat core)**.** *An* unsat core *of a CNF formula F is an unsatisfiable subset of F.*

Definition 2.4 (model extension)**.** *Let C be a CNF formula, F a satisfiable subset of C, and π a model of F. The* model extension *of F w.r.t. π and C is the set $E = \{c \in C \,|\, \pi \models c\}$. Note that $F \subseteq E \subseteq C$ and E is satisfiable (π is its model).*

2.2 *Minimal and Maximal Sets over a Monotone Predicate*

This section introduces the notion of minimal and maximal sets over a monotone predicate [Marques-Silva et al., 2013b, 2017]. In general, we are given a finite set $C = \{c_1, c_2, \ldots, c_n\}$ of elements together with a *monotone* predicate $\mathbf{P} : \mathcal{P}(\mathcal{C}) \to \{1, 0\}$. The monotonicity and the minimal and maximal sets are defined as follows.

Definition 2.5 (monotone predicate)**.** *A predicate $\mathbf{P} : \mathcal{P}(C) \to \{1, 0\}$ over a reference set C is* monotone *iff whenever $\mathbf{P}(N) = 1$, with $N \subseteq C$, then $\mathbf{P}(N') = 1$ for every $N \subseteq N' \subseteq C$.*

Definition 2.6 ($\mathbf{P}_1$-minimal, MSMP)**.** *A set N is a $\mathbf{P}_1$-minimal subset of C iff $N \subseteq C$, $\mathbf{P}(N) = 1$, and $\mathbf{P}(N') = 0$ for every $N' \subsetneq N$. We also refer to a $\mathbf{P}_1$-minimal subset of C as to a* minimal subset of C over a monotone predicate $\mathbf{P}$*, shortly* MSMP.

Definition 2.7 ($\mathbf{P}_0$-maximal)**.** *A set N is a $\mathbf{P}_0$-maximal subset of C iff $N \subseteq C$, $\mathbf{P}(N) = 0$, and $\mathbf{P}(N') = 1$ for every $N \subsetneq N' \subseteq C$. We also refer to a $\mathbf{P}_0$-maximal subset of C as to a* maximal subset of C over a monotone predicate $\mathbf{P}$.

Note that the maximality (minimality) concept is the set maximality (minimality), not maximum (minimum) cardinality as e.g. in the MaxSAT problem. Consequently, there can be multiple $\mathbf{P}_1$-minimal and $\mathbf{P}_0$-maximal subsets of C with different cardinalities. The maximum number of $\mathbf{P}_1$-minimal (and also $\mathbf{P}_0$-maximal) subsets of C, is bounded due to Sperner's theorem [Sperner, 1928] to $\binom{n}{\lfloor n/2 \rfloor}$ where $n = |C|$.[1]

[1] Intuitively, this is the width of the widest row of the power-set $\mathcal{P}(C)$ of C, i.e., the maximum number of pairwise incomparable subsets of C w.r.t. the subset inclusion.

Note that to prove that a set N is a $\mathbf{P}_1$-minimal/$\mathbf{P}_0$-maximal subset of C, one does not have to consider all subsets/supersets of N as witnessed in Observations 2.3 and 2.4.

Observation 2.3. *A set N is a $\mathbf{P}_1$-minimal subset of C iff $N \subseteq C$, $\mathbf{P}(N) = 1$, and $\mathbf{P}(N \backslash \{c\}) = 0$ for every $c \in N$.*

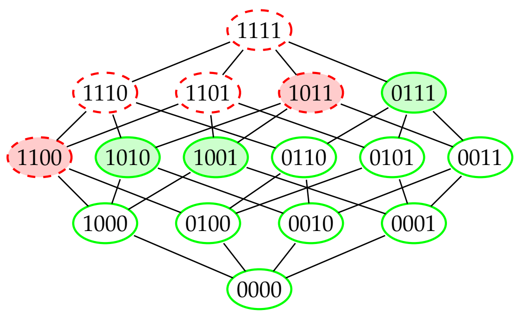


Figure 2.1: Illustration of the power-set $\mathcal{P}(C)$ from Example 2.2. We encode individual subsets of C as bitvectors, e.g., $\{c_1, c_2, c_4\}$ is encoded as 1101. Satisfiable and unsatisfiable subsets are drawn in green and red, respectively. MUSes and MSSes are filled with a background color.
Note that due to the monotonicity of the satisfiability predicate, there is a borderline between satisfiable and unsatisfiable subsets. The borderline can be defined exactly by the maximal satisfiable or minimal unsatisfiable subsets. This holds for all instances of minimal sets over a monotone predicate.

Proof. $\Rightarrow$: Directly by Definition 2.6.
$\Leftarrow$: Assume that $\mathbf{P}(N\backslash\{c\}) = 0$ for every $c \in N$, and that there exists N' such that $N' \subsetneq N$ with $\mathbf{P}(N') = 1$ (i.e., N is not $\mathbf{P}_1$-minimal). Since $N' \subsetneq N$, then $N' \subseteq N\backslash\{c\}$ for some $c \in N$. By the monotonicity (Definition 2.5), since $\mathbf{P}(N') = 1$ then $\mathbf{P}(N\backslash\{c\}) = 1$ (contradiction). □

Observation 2.4. *A set N is a $\mathbf{P}_0$-maximal subset of C iff $N \subseteq C$, $\mathbf{P}(N) = 0$, and $\mathbf{P}(N \cup \{c\}) = 1$ for every $c \in C\backslash N$.*

Proof. Dually to the proof of Observation 2.3. □

We illustrate the concepts of a monotone predicate and $\mathbf{P}_1$-minimal and $\mathbf{P}_0$-maximal subsets on several examples.

Example 2.1. *Assume that C is a finite set of positive integers, and let $\mathbf{P}(N) = 1$ iff $\sum_{c \in N} c > 100$ for every $N \subseteq C$. Here, $\mathbf{P}$ is monotone on C since C contains only positive integers. However, if we assume that C can contain both positive and negative integers, then $\mathbf{P}$ is not monotone on C.*

Example 2.2. *Assume that the set C is a set of Boolean clauses, i.e., a CNF formula, and the predicate $\mathbf{P}$ is defined as $\mathbf{P}(N) = 1$ iff the subset N of C is unsatisfiable. In such case, the $\mathbf{P}_1$-minimal subsets are the* minimal unsatisfiable subsets (MUSes) *of C, and the $\mathbf{P}_0$-maximal subsets are the* maximal satisfiable subsets (MSSes) *of C. In particular, assume that C contains four clauses: $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$. There are two MUSes: $\{c_1, c_2\}$, $\{c_1, c_3, c_4\}$, and three MSSes: $\{c_1, c_4\}$, $\{c_1, c_3\}$, $\{c_2, c_3, c_4\}$. The situation is illustrated in Figure 2.1.*

Example 2.3. *Assume that C is a set of Boolean clauses and $\mathbf{P}$ is defined as $\mathbf{P}(N) = 1$ iff the set $C\backslash N$ is satisfiable. In this case, the $\mathbf{P}_1$-minimal subsets are the* minimal correction subsets *(MCSes) of C, i.e., the minimal sets of clauses that need to be removed from C to make it satisfiable. In particular, if C contains the four clauses as in Example 2.2, then there are three MCSes: $\{c_2, c_3\}$, $\{c_2, c_4\}$, $\{c_1\}$.*

Note that in Examples 2.2 and 2.3 we use the same set C, and that the $\mathbf{P}_1$-minimal subsets from Example 2.3 (i.e., MCSes) are exactly the complements of the $\mathbf{P}_0$-maximal subsets from Example 2.2 (i.e., MSSes). This is not a coincidence; it holds in general that maximal subsets of a set C over a monotone predicate can be expressed as minimal subsets of C over another monotone predicate:

Observation 2.5. *Let C be a finite set of elements and $\mathbf{P}$ a monotone predicate over C. Furthermore, let $\mathbf{Q}$ be a predicate over C defined as $\mathbf{Q}(C\backslash N) = 1$ iff $\mathbf{P}(N) = 0$ for every $N \subseteq C$. Then $\mathbf{Q}$ is monotone, and a set N is a $\mathbf{P}_0$-maximal subset of C iff $C\backslash N$ is a $\mathbf{Q}_1$-minimal subset of C.*

Proof. We start with the monotonicity of $\mathbf{Q}$. By contradiction, assume that there are N, N' such that $N \subseteq N' \subseteq C$, $\mathbf{Q}(N) = 1$ and $\mathbf{Q}(N') = 0$. By the definition of $\mathbf{Q}$, we have $\mathbf{P}(C\backslash N) = 0$ and $\mathbf{P}(C\backslash N') = 1$, and since $N \subseteq N'$ we have $(C\backslash N') \subseteq (C\backslash N)$ which contradicts that $\mathbf{P}$ is monotone.
As for the $\mathbf{P}_0$-maximal and $\mathbf{Q}_1$-minimal equivalence, assume a subset N of C such that N is $\mathbf{P}_0$-maximal, however, $C\backslash N$ is not $\mathbf{Q}_1$-minimal. Consequently, either $\mathbf{Q}(C\backslash N) = 0$ or there exists N' such that $(C\backslash N') \subsetneq (C\backslash N)$ and $\mathbf{Q}(C\backslash N') = 1$. The former case contradicts that $\mathbf{P}(N) = 0$. In the latter case we have $\mathbf{P}(N') = 0$ which contradicts that N $\mathbf{P}_0$-maximal since $N \subsetneq N'$. Dually for the other direction.

□

Consequently, the concept of minimal subsets that satisfy a monotone predicate allows us to describe the same class of problems as the concept of maximal subsets that do not satisfy a monotone predicate. Thus, we could restrict ourselves to using only one of the two concepts. In fact, note that Marques-Silva et al. who pioneered the study on the general concept of minimal/maximal sets over a monotone predicate [Marques-Silva et al., 2013b, Janota and Marques-Silva, 2016, Marques-Silva et al., 2017] used only the term of *minimal elements* in their definitions (their definition of a minimal set over a monotone predicate (MSMP) corresponds to our definition of a $\mathbf{P}_1$-minimal subset). However, in some situations, it is more natural to speak about maximal subsets, and in other situations about minimal subsets. Hence, we use both the terms in this thesis.

In the rest of this chapter, we summarize several well-known properties, facts, and techniques related to minimal and maximal sets over a monotone predicate that are used throughout the thesis. Especially, we focus on concepts related to *enumeration* of minimal and maximal sets over a monotone predicate.

2.3 *General Properties of Minimal and Maximal Sets over a Monotone Predicate*

Hereafter, let us use C and $\mathbf{P}$ to denote the reference set and the monotone predicate of interest, respectively. There is a well-known relationship between $\mathbf{P}_1$-minimal subsets and the complements of $\mathbf{P}_0$-maximal subsets of C. The relationship is defined in terms of *minimal hitting sets*. Given a collection Ω of sets, a *hitting set* H of Ω is a set such that $\forall S \in \Omega : H \cap S \neq \emptyset$. A hitting set is called *minimal* if none of its proper subsets is a hitting set. The following Observation 2.6, called *minimal hitting set duality*, was first pointed out by Reiter [Reiter, 1987] and by de Kleer and Williams [de Kleer and Williams, 1987].

Observation 2.6 (minimal hitting set duality)**.** *A set N is a $\mathbf{P}_1$-minimal subset of C if and only if N is a minimal hitting set of $\mathcal{X}$, where $\mathcal{X}$ is the set of complements of all $\mathbf{P}_0$-maximal subsets of C, i.e., $\mathcal{X} = \{X \mid C \backslash X$ is $\mathbf{P}_0$-maximal$\}$. Similarly, a set N is a $\mathbf{P}_0$-maximal subset of C if and only if $C \backslash N$ is a minimal hitting set of the set of all $\mathbf{P}_1$-minimal subsets of C.*

Intuitively, due to the monotonicity of $\mathbf{P}$, a $\mathbf{P}_0$-maximal subset N of C cannot contain any $\mathbf{P}_1$-minimal subset of C, and hence the complement of N has to hit every $\mathbf{P}_1$-minimal subset of C, and vice versa. Another widely known concepts are *critical* and *conflicting* elements (sometimes also called *transition* elements [Belov and Marques-Silva, 2011, Marques-Silva et al., 2013b]).

Definition 2.8 (critical element)**.** *Let N be a subset of C such that $\mathbf{P}(N) = 1$. An element $c \in N$ is* critical *for N if $\mathbf{P}(N \backslash \{c\}) = 0$.*

Note that if c is a critical element for C then c has to be contained in every subset C' of C such that $\mathbf{P}(C') = 1$ and especially in every $\mathbf{P}_1$-minimal subset of C. The opposite does not have to hold; if $N \subseteq C$ and c is critical for N then c is not necessarily critical for C, i.e., there can be $\mathbf{P}_1$-minimal subsets of C that do not contain c. Furthermore, note that Observation 2.7 holds.

Observation 2.7. *A subset N of C is a $\mathbf{P}_1$-minimal subset of C iff every $c \in N$ is critical for N.*

Proof. Immediately from Observation 2.3. □

Definition 2.9 (conflicting element)**.** *Let N be a subset of C such that $\mathbf{P}(N) = 0$. An element $c \in C \backslash N$ is* conflicting *for N if $\mathbf{P}(N \cup \{c\}) = 1$.*

Similarly to the case of critical elements, note that if c is conflicting for a set N with $\mathbf{P}(N) = 0$, then c is conflicting for every superset N' of N with $\mathbf{P}(N') = 0$. Furthermore, Observation 2.8 holds.

Observation 2.8. *A subset N of C is a $\mathbf{P}_0$-maximal subset of C iff every $c \in C \backslash N$ is conflicting for N.*

Proof. Immediately from Observation 2.4. □

2.3.1 Explored and Unexplored Subsets

In the thesis, we present several algorithms for enumeration of $\mathbf{P}_1$-minimal and/or $\mathbf{P}_0$-maximal subsets of C. The algorithms during their computation gradually *explore* the *status* of subsets of C, i.e., determine whether the predicate $\mathbf{P}$ holds for the individual subsets or not. A subset N of C is *explored* by an algorithm if the algorithm has already determined whether $\mathbf{P}(N) = 0$ or $\mathbf{P}(N) = 1$; otherwise, N is *unexplored*. To avoid repeatedly checking already explored subsets of C, the algorithms maintain a set `Unexplored` that stores all the unexplored subsets (and thus implicitly also the explored subsets). Furthermore, algorithms (not just ours) we describe in the thesis obey the following four rules, R1, R2, R3, and R4, for a manipulation with the set `Unexplored`:

R1: At the start of the computation (of the algorithm), `Unexplored` $= \mathcal{P}(C)$, i.e., all subsets of C are unexplored.

R2: Except the initialization step (R1), no element can be added to `Unexplored` (i.e., everything that becomes explored stays explored).

R3: Whenever a set S with $\mathbf{P}(S) = 0$ is removed from `Unexplored`, then all subsets of S are also removed from `Unexplored` (i.e., if S is explored then all subsets of S are also explored). This rule is based on the monotonicity of the predicate $\mathbf{P}$; since $\mathbf{P}(S) = 0$, then also $\mathbf{P}(S') = 0$ for every $S' \subseteq S$.

R4: Dually to the third rule, if a set U such that $\mathbf{P}(U) = 1$ is removed from `Unexplored`, then all supersets of U are also removed from `Unexplored` (i.e., if U is explored then all supersets of U are also explored).

Based on the above rules on the manipulation with `Unexplored`, we can state several additional invariant properties of `Unexplored`.

Observation 2.9. *If $N \in$ `Unexplored` and $\mathbf{P}(N) = 0$, then for every superset N' of N such that $\mathbf{P}(N') = 0$ it holds that $N' \in$ `Unexplored`.*

Proof. By contradiction, assume a superset N' of N such that $\mathbf{P}(N') = 0$ and $N' \notin$ `Unexplored`. Due to rule R3, if N' is explored then N is also explored, i.e., $N \notin$ `Unexplored`. □

Observation 2.10. *If $N \in$ `Unexplored` and $\mathbf{P}(N) = 1$, then for every subset N' of N such that $\mathbf{P}(N') = 1$ it holds that $N' \in$ `Unexplored`.*

Proof. Dually to the proof of Observation 2.9. □

Observation 2.11. *Every $\mathbf{P}_1$-minimal and every $\mathbf{P}_0$-maximal subset of C is explored iff all subsets of C are explored (i.e., `Unexplored` $= \emptyset$).*

Proof. $\Rightarrow$: By Definition 2.7, for every subset N of C such that $\mathbf{P}(N) = 0$ there exists a $\mathbf{P}_0$-maximal subset N' of C such that $N \subseteq N'$. Dually, by Definition 2.6, for every subset N of C such that $\mathbf{P}(N) = 1$ there exists a $\mathbf{P}_1$-minimal subset N' of C such that $N' \subseteq N$. All $\mathbf{P}_0$-maximal and $\mathbf{P}_1$-minimal subsets of C are explored and, thus, based on rules R3 and R4, all their subsets and supersets, respectively, are also explored.
$\Leftarrow$: Every $\mathbf{P}_0$-maximal and every $\mathbf{P}_1$-minimal subset of C is a subset of C. □

Another invariant properties of `Unexplored` concerns *minimal* and *maximal* unexplored subsets which are defined as follows:

Definition 2.10 (minimal unexplored subset)**.** *A subset N of C is a* minimal unexplored subset *(of C) iff N is unexplored and $\forall c \in N$ the set $N \setminus \{c\}$ is explored.*

Definition 2.11 (maximal unexplored subset)**.** *A subset N of C is a* maximal unexplored subset *(of C) iff N is unexplored and $\forall c \in C \setminus N$ the set $N \cup \{c\}$ is explored.*

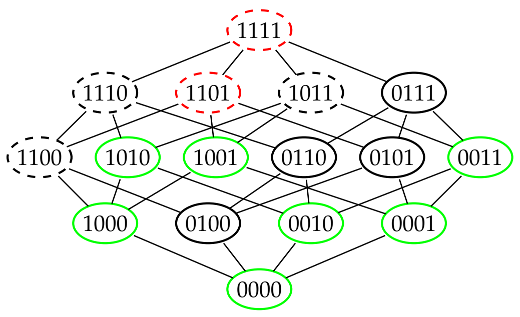


Figure 2.2: Illustration of the explored and unexplored subsets from Example 2.4. We encode individual subsets of C as bitvectors, e.g., $\{c_1, c_2, c_4\}$ is encoded as 1101. There are two explored unsatisfiable subsets (red with dashed border), seven explored satisfiable subsets (green with solid border), and seven unexplored subsets (drawn in black). Three of the unexplored subsets are unsatisfiable (black with dashed border) and four of the unexplored subsets are satisfiable (black with solid border). There are three maximal unexplored subsets: $\{c_1, c_2, c_3\}$, $\{c_1, c_3, c_4\}$ and $\{c_2, c_3, c_4\}$, and two minimal unexplored subset: $\{c_2\}$ and $\{c_1, c_3, c_4\}$.

Observation 2.12. *Every maximal unexplored subset N such that $\mathbf{P}(N) = 0$ is a $\mathbf{P}_0$-maximal subset of C.*

Proof. By contradiction, assume that there is an element $c \in C \backslash N$ such that $\mathbf{P}(N \cup \{c\}) = 0$. Since N is a maximal unexplored subset, it holds that $N \cup \{c\}$ is explored. However, due to rule R3, all subsets of $N \cup \{c\}$ (including N) are also explored (which contradicts that N is unexplored). □

Observation 2.13. *Every minimal unexplored subset N such that $\mathbf{P}(N) = 1$ is a $\mathbf{P}_1$-minimal subset of C.*

Proof. Dually to Observation 2.12. □

Example 2.4. *To illustrate the concepts, assume that C is a set of Boolean clauses and $\mathbf{P}$ holds for a subset N of C (i.e., $\mathbf{P}(N) = 1$) iff N is unsatisfiable. Thus, $\mathbf{P}_1$-minimal subsets of C are minimal unsatisfiable subsets of C and $\mathbf{P}_0$-maximal subsets of C are maximal satisfiable subsets of C. Let $C = \{c_1, c_2, c_3, c_4\}$ with $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$ (i.e., the same clauses as in Example 2.2). Figure 2.2 shows a possible state of exploration of the power-set of C.*

Two following two observations about the set `Unexplored` allow us to *mine* critical and conflicting elements for some unexplored subsets.

Observation 2.14. *Let N be an unexplored subset such that $\mathbf{P}(N) = 1$. Then for every $c \in N$ such that $N \backslash \{c\} \notin$ `Unexplored` it holds that c is critical for N, i.e., $\mathbf{P}(N \backslash \{c\}) = 0$.*

Proof. Assume that $\mathbf{P}(N \backslash \{c\}) = 1$. By rule R4, if $N \backslash \{c\}$ is explored then N is also explored (contradiction). □

Observation 2.15. *Let N be an unexplored subset such that $\mathbf{P}(N) = 0$. Then for every $c \in C \backslash N$ such that $N \cup \{c\} \notin$ `Unexplored` it holds that c is conflicting for N, i.e., $\mathbf{P}(N \cup \{c\}) = 1$.*

Proof. Assume that $\mathbf{P}(N \cup \{c\}) = 0$. By rule R3, if $N \cup \{c\}$ is explored then N is also explored (contradiction). □

Definition 2.12 (minable critical)**.** *Let N be an unexplored subset with $\mathbf{P}(N) = 1$, and let c be a critical element for N. The element c is* minable critical *for N if $N \backslash \{c\} \notin$ `Unexplored`.*

Definition 2.13 (minable conflicting). *Let N be an unexplored subset with $\mathbf{P}(N) = 0$, and let c be a conflicting element for N. The element c is* minable *conflicting for N if $N \cup \{c\} \notin$* `Unexplored`.

Example 2.5. *For instance, examine the set $N = \{c_1, c_2, c_3\}$ with $\mathbf{P}(N) = 1$ in Example 2.4. We can see that c_2 is critical for N since $N \backslash \{c_2\}$ is explored and thus satisfiable.*

Given an unexplored N such that $\mathbf{P}(N) = 1$, the ability to mine critical elements for N is very beneficial in a situation where we want to find a $\mathbf{P}_1$-minimal subset M of N. Since every critical element of N has to be contained in every $\mathbf{P}_1$-minimal subset of N, prior knowledge of a set of critical elements can significantly speed up the extraction of M. Dually, given an unexplored N such that $\mathbf{P}(N) = 0$, prior knowledge of elements that are conflicting for N can be very helpful if we want to identify a $\mathbf{P}_0$-maximal subset N_{mss} of C such that $N \subseteq N_{mss}$. The role of critical (conflicting) elements during extracting $\mathbf{P}_1$-minimal ($\mathbf{P}_0$-maximal) subsets is described in more detail in Section 2.3.3.

2.3.2 *Representation of Unexplored Subsets*

There are various ways of representing the unexplored subsets. Perhaps the most straightforward way is to explicitly hold the set of all unexplored subsets and/or the set of all explored subsets. However, due to the exponential growth of $\mathcal{P}(C)$ w.r.t. $|C|$, an explicit representation of unexplored subsets is impractical for larger input sets.

In all our algorithms we present in this thesis, we adopt a symbolic representation that was originally proposed, in the context of MUS enumeration, by Liffiton et al. [Liffiton and Malik, 2013, Previti and Marques-Silva, 2013, Liffiton et al., 2016] and used in their algorithm MARCO. The representation exploits the well-known isomorphism between finite power-sets and Boolean algebras. Given a set $C = \{c_1, \ldots, c_n\}$ of elements, we introduce a set $X = \{x_1, \ldots, x_n\}$ of Boolean variables. Note that every valuation of X one-to-one maps to a subset of C. To represent the unexplored subsets, we maintain two Boolean formulas in CNF, map^+ and map^-, over X such that each model of the conjunction $map^+ \wedge map^-$ corresponds to an unexplored subset of C and vice versa. The formulas are maintained as follows:

- Initially $map^+ = map^- = True$ since the whole $\mathcal{P}(C)$ is unexplored (according to the rule R1).
- To mark a set $N \subseteq C$ such that $\mathbf{P}(N) = 0$ and all its subsets as explored (according to the rule R3), we add the clause $\bigvee_{c_i \notin N} x_i$ to map^+.
- Symmetrically, to mark a set $N \subseteq C$ such that $\mathbf{P}(N) = 1$ and all its supersets as explored (according to the rule R4), we add the clause $\bigvee_{c_i \in N} \neg x_i$ to map^-.

Example 2.6. *Assume that we are given the set C and the predicate* $\mathbf{P}$ *from Example 2.4, i.e., C contains 4 Boolean clauses: $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$, and* $\mathbf{P}$ *is the standard Boolean unsatisfiability. If all elements of $\mathcal{P}(C)$ are unexplored then $map^+ = map^- = True$. If $\{c_1, c_2, c_4\}$ is found to be unsatisfiable and $\{c_1, c_3\}, \{c_1, c_4\}, \{c_3, c_4\}$ to be satisfiable, then $map^- = (\neg x_1 \vee \neg x_2 \vee \neg x_4)$ and $map^+ = (x_2 \vee x_4) \wedge (x_2 \vee x_3) \wedge (x_1 \vee x_2)$. Note that this state of exploration is the same as the one used in Example 2.4 and illustrated in Figure 2.2.*

To get an arbitrary unexplored subset, we ask a SAT solver for a model of $map^+ \wedge map^-$. To verify if a subset N of C is unexplored, we check if the valuation of X that corresponds to N is a model of $map^+ \wedge map^-$. Furthermore, in our algorithms, we need to be able to obtain the following specific unexplored subsets.

First, for a given $S \subseteq C$, we need to be able to obtain a maximal unexplored subset of S, i.e., a subset N of S such that $\forall c \in S \backslash N$ the set $N \cup \{c\}$ is explored. Similarly, we need to be able to obtain minimal unexplored subsets of S. As was already noted in [Liffiton et al., 2016], maximal and minimal unexplored subsets correspond to maximal and minimal models of $map^+ \wedge map^-$. To get a maximal unexplored subset of S, we instruct the SAT solver to fix the truth assignment of the variables $\{x_i | c_i \in C \backslash S\}$ to *False* and ask for a maximal model of $map^+ \wedge map^-$; similarly for the minimal case.

Second, for a given minimal *unexplored* subset N_0, we need to be able to produce a maximal unexplored subset N_k such that $N_k \supseteq N_0$. This can be achieved by instructing the SAT solver to fix the truth assignment of the variables $\{x_i | c_i \in N_0\}$ to *True* and asking for a maximal model of $map^+ \wedge map^-$. A straightforward improvement to this approach can be done if we note that map^+ is satisfied by every variable assignment that sets the variables $\{x_i | c_i \in N_0\}$ to *True* (since N_0 is unexplored and map^+ intuitively only requires a presence of elements of C). Therefore, it is enough to fix $\{x_i | c_i \in N_0\}$ to *True* and ask just for a maximal model of map^-. Note that this query can be solved in polynomial time. In particular, one can start with the satisfying assignment that corresponds to N_0, and then attempt to switch one by one the variables that correspond to $C \backslash N_0$ from *False* to *True* and keep only the changes that preserve the satisfaction of map^-.

Finally, given a subset S of C, we need to be able to obtain an unexplored subset that contains the whole S. To do that, we simply instruct the SAT solver to fix the values of the variables $\{x_i | c_i \in S\}$ to *True* and ask for a model of $map^+ \wedge map^-$.

For the implementation, we use the miniSAT [Eén and Sörensson, 2003] solver, which fulfills our requirements, i.e., it allows fixing truth assignments to variables and can produce minimal and maximal models.[2] We use miniSAT in an incremental manner, i.e., we hold a single instance of miniSAT during the whole computation of an algorithm and incrementally add clauses to the formula $map^+ \wedge map^-$. Furthermore, the formula $map^+ \wedge map^-$ can be internally simplified by miniSAT when possible. Let us note that Liffiton

[2] The minimal and maximal models are enforced in miniSAT by setting the default polarity (values) of variables at decision points during the solving to *False* and *True*, respectively.

et al. [Liffiton et al., 2016], who first proposed this kind of symbolic representation, also incrementally use miniSAT in the above-described manner.

2.3.3 *Shrink and Grow*

We now define two procedures, shrink and grow, that are used as subroutines of many existing algorithms for enumeration of $\mathbf{P}_1$-minimal and $\mathbf{P}_0$-maximal subsets (including the algorithms presented in this thesis).

- shrink(N, crits) takes as an input a subset N of C such that $\mathbf{P}(N) = 1$ together with a set *crits* of elements that are critical for N. The output is a $\mathbf{P}_1$-minimal subset N' of C such that $N' \subseteq N \subseteq C$. Moreover, note that if N is unexplored, then N' is also unexplored (Observation 2.10).
- grow$(N, \mathit{conflicts})$ takes as an input a subset N of C such that $\mathbf{P}(N) = 0$ together with a set *conflicts* of elements that are conflicting for N. The output is a $\mathbf{P}_0$-maximal subset N' of C such that $N \subseteq N' \subseteq C$. Moreover, note that if N is unexplored, then N' is also unexplored (Observation 2.9).

We refer to the execution of shrink(N, crits) as to *shrinking* N into a $\mathbf{P}_1$-minimal subset of N. Similarly, the execution of grow$(N, \mathit{conflicts})$ is called *growing* N into a $\mathbf{P}_0$-maximal subset of C.

In general, the shrinking and the growing is implemented via a single $\mathbf{P}_1$-minimal and $\mathbf{P}_0$-maximal subset extraction procedure, respectively. There have been proposed a plethora of such extractors, e.g., [Nadel et al., 2014, Belov and Marques-Silva, 2012, Belov et al., 2014, Bacchus and Katsirelos, 2015, Ghassabani et al., 2016, Marques-Silva et al., 2013a, Mencía et al., 2016, Felfernig et al., 2012, Bailey and Stuckey, 2005]. Most of the approaches are domain specific, i.e., tailored for a particular type of the set C and the predicate $\mathbf{P}$ (e.g., Boolean clauses and Boolean unsatisfiability). However, some of the algorithms are domain agnostic, i.e., they can be applied for an arbitrary type of C and $\mathbf{P}$ (as long as C is finite and $\mathbf{P}$ monotone). To at least illustrate how the shrinking and growing can be performed, let us here present two simple domain agnostic procedures (one for growing and one for shrinking). More advanced domain specific solutions are discussed later in the thesis.

Shrinking In Algorithm 2.1, we show a *deletion-based* shrinking approach [Chinneck and Dravnieks, 1991, Bakker et al., 1993]. The algorithm iteratively maintains a subset N of C such that $\mathbf{P}(N) = 1$ and a set *crits* of elements that are critical for N. In each iteration, the algorithm picks an element $c \in N \backslash \mathit{crits}$ and checks if the predicate $\mathbf{P}$ holds for $N \backslash \{c\}$. If $\mathbf{P}(N \backslash \{c\}) = 0$, then c is critical for N, and hence it is added to *crits*. Otherwise, if $\mathbf{P}(N \backslash \{c\}) = 1$, the element c is removed from N. The algorithm terminates once $N = \mathit{crits}$. The invariant of the algorithm is that $\mathit{crits} \subseteq N$ and every $c \in \mathit{crits}$ is critical

```
input : a subset $N$ of $C$ such that $\mathbf{P}(N) = 1$
input : a set $crits$ of elements that are critical for $N$
output: a $\mathbf{P}_1$-minimal subset $N'$ of $C$ such that $N' \subseteq N \subseteq C$
1 while $N \setminus crits \neq \emptyset$ do
2     $c \leftarrow$ pick $c \in N \setminus crits$
3     if $\mathbf{P}(N \setminus \{c\}) = 1$ then
4         $N \leftarrow N \setminus \{c\}$
5     else
6         $crits \leftarrow crits \cup \{c\}$
7 return $N$
```

Algorithm 2.1: Domain agnostic shrinking (one of existing approaches).

for N. Thus, when $N = crits$, it is guaranteed that N is a $\mathbf{P}_1$-minimal subset of C (Observation 2.7). Also, note that in every iteration, either N is reduced or *crits* is enlarged, hence it is guaranteed that the computation terminates.

In total, the algorithm performs $|N_0| - |crits_0|$ predicate checks, where N_0 and $crits_0$ are the initial input values. Contemporary domain specific algorithms (e.g., [Belov and Marques-Silva, 2012, Nadel et al., 2014, Belov et al., 2014, Ghassabani et al., 2016]) based on Algorithm 2.1 are often able to significantly reduce the number of performed predicate checks. For example, if C is a set of Boolean clauses and the predicate checks are carried out by a SAT solver, Algorithm 2.1 can be improved by utilizing *unsat cores* provided by a SAT solver.

Let us note that there have been proposed several other domain agnostic solutions that conceptually differ from Algorithm 2.1. For example, a *constructive* approach [de Siqueira N. and Puget, 1988] starts with the empty set and iteratively adds elements from N_0 to the empty set to construct a $\mathbf{P}_1$-minimal subset of N_0. The constructive approach performs $\mathcal{O}(|N_0| \times k)$ predicate checks where k is the cardinality of the largest $\mathbf{P}_1$-minimal subset of N_0. Another approach, called *dichotomic* [Hemery et al., 2006], is based on binary search and performs $\mathcal{O}(k \times \log |N_0|)$ predicate checks. The main takeaway for the reader is that the cardinality of the set that is being shrunk significantly affects the complexity of the shrinking. Also, prior knowledge of a set of critical constraints for N_0 can significantly speed up the shrinking.

Growing A growing procedure [Bailey and Stuckey, 2005] that works in a dual way to the deletion-based shrinking approach is shown in Algorithm 2.2. The algorithm iteratively maintains a subset N of C such that $\mathbf{P}(N) = 0$ and a set *conflicts* of elements that are conflicting for N. In each iteration, the algorithm picks an element $c \in (C \setminus N) \setminus conflicts$ and checks if the predicate $\mathbf{P}$ holds for $N \cup \{c\}$. If $\mathbf{P}(N \cup \{c\}) = 1$, then c is conflicting for N, and hence it is added to *conflicts*. Otherwise, if $\mathbf{P}(N \cup \{c\}) = 0$, then the element c is added to N. The invariant of the algorithm is that $conflicts \subseteq C \setminus N$ and that

```
input : a subset $N$ of $C$ such that $\mathbf{P}(N) = 0$
input : a set $conflicts$ of elements that are conflicting for $N$
output: a $\mathbf{P}_0$-maximal subset $N'$ of $C$ such that $N \subseteq N'$
1 while $conflicts \neq C \backslash N$ do
2     $c \leftarrow$ pick $c \in (C \backslash N) \backslash conflicts$
3     if $\mathbf{P}(N \cup \{c\}) = 0$ then
4         $N \leftarrow N \cup \{c\}$
5     else
6         $conflicts \leftarrow conflicts \cup \{c\}$
7 return $N$
```

Algorithm 2.2: Domain agnostic growing (one of existing approaches).

every $c \in conflicts$ is conflicting for N. Hence, once $conflicts = C \backslash N$, it is guaranteed that N is a $\mathbf{P}_0$-maximal subset of C (Observation 2.8). The termination is guaranteed since in every iteration either $C \backslash N$ is reduced or *conflicts* is enlarged. The total number of performed predicate checks is $|C| - (|N_0| + conflicts_0)$, where N_0 and $conflicts_0$ are the initial input values.

2.4 *Related Work*

There have been studied a plethora of computational problems that can be formulated as computing minimal sets subject to a monotone predicate, and it is hard to identify the very first work on this topic. For instance, de Kleer and Williams [de Kleer and Williams, 1987] and Reiter [Reiter, 1987] studied the problems of identifying *minimal diagnoses* and *minimal conflicts* in the context of diagnosing misbehaving systems. Another example is the work by Ben-Eliyahu and Dechter [Ben-Eliyahu and Dechter, 1993] who focused on computing minimal models of a propositional formula, or the work by Chinneck and Dravnieks [Chinneck and Dravnieks, 1991] on locating minimal infeasible constraint sets in linear programs. Yet other examples include, e.g., the computation of minimal unsatisfiable and maximal satisfiable subsets [Bailey and Stuckey, 2005], a maximal autarky [Marques-Silva et al., 2014], or minimal independent supports [Ivrii et al., 2016].

To the best of our knowledge, Marques-Silva, Janota, and Belov were the first who noted that there are so many computational problems with the same underlying, monotone, structure [Marques-Silva et al., 2013b]. In particular, they introduced the general concept of *minimal sets over a monotone predicate (MSMPs)*, shown several computational problems that can be cast as computing MSMPs, and proposed an algorithm for computing a single MSMP. Subsequently, Marques-Silva and Janota studied the query complexity of finding minimal sets over a monotone predicate [Janota and Marques-Silva, 2016]. Moreover, Marques-Silva, Janota, and Mencía [Marques-Silva et al., 2017] provided a long list of particular instances of the general problem of identifying MSMPs that arise in the context of propositional formulae.

There have been also many studies on various decision and functional problems related to MSMPs, e.g., extraction of a single or all MSMPs, extraction of the smallest MSMP, or deciding whether there is an MSMP that contains a particular element (of the reference set C). Some of these studies focused on particular instances of MSMPs (e.g., minimal models), and some apply to the general MSMP settings. We postpone the discussion about studies that are tightly connected to our work to the appropriate following chapters.

Part I

Domain Agnostic MUS Enumeration

3
The Role and Applications of Domain Agnostic MUS Enumeration Algorithms

In the past decades, there emerged a plethora of various instances of MSMPs, e.g., minimal independent supports [Ivrii et al., 2016], minimal inconsistent subsets [Barnat et al., 2016], minimal inductive validity cores [Ghassabani et al., 2016], minimal models [Ben-Eliyahu and Dechter, 1993], or minimal correction subsets [Marques-Silva et al., 2013a]. It is often the case that the input set C is a set of some *constraints* and the monotone predicate $\mathbf{P}$ is a *constraint unsatisfiability*. In such case, the $\mathbf{P}_1$-minimal subsets of C are called *minimal unsatisfiable subsets (MUSes)* of C. For example, C can be a set of Boolean, SMT, or LTL formulae, and $\mathbf{P}$ the standard Boolean, SMT, or LTL, unsatisfiability, respectively. In this part of the thesis, we focus on *domain agnostic* MUS enumeration algorithms. A domain agnostic MUS enumeration algorithm takes as an input a finite set C of arbitrary constraints, and it enumerates/identifies all MUSes of C.

A natural role of domain agnostic MUS enumeration algorithms is to serve as ready-to-use solutions for any constraint domain where a new application of MUSes arises. Of course, an algorithm that is tailored for an MUS enumeration in a particular constraint domain will probably perform better than a domain agnostic MUS enumeration algorithm, since domain agnostic algorithms cannot fully exploit properties of particular constraint domains. However, development of such a domain specific algorithm takes some time. Consequently, it is often the case that a domain agnostic algorithm is used in the meantime before a domain specific solution is developed. Moreover, it is often the case that the domain specific algorithms are based on existing domain agnostic solutions.

To illustrate the variety of constraint domains where domain agnostic MUS enumerators can be applied, we first describe in Chapter 3 (this chapter) several particular applications of MUSes. Then, in Chapter 4, we present our two domain agnostic MUS enumeration algorithms: TOME [Bendík et al., 2016b] and ReMUS [Bendík et al., 2018b].

Finally, let us note that the domain agnostic MUS enumeration algorithms we discuss here can be in fact used for any instance of the general MSMP enumeration problem, i.e., the input set C can contain

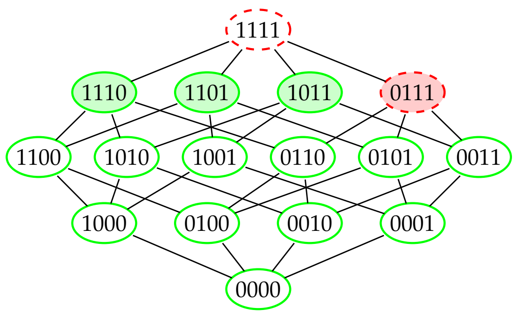


Figure 3.1: Illustration of the requirements analysis example. We encode individual subsets as bit-vectors, e.g., $\{\varphi_2, \varphi_3, \varphi_4\}$ is encoded as 0111. The subsets with solid green border are consistent, whereas the subsets with red dashed border are inconsistent. The maximal consistent and minimal inconsistent subsets are filled with a background color.

arbitrary elements and $\mathbf{P} : \mathcal{P}(C) \rightarrow \{1,0\}$ can be an arbitrary monotone predicate. Yet, it is unclear how efficient would the MUS enumeration algorithms be in the general MSMP setting. In particular, the design of the MUS enumeration algorithms is based on several assumptions that are specific for unsatisfiable constraint systems. For instance, the predicate checks, i.e., unsatisfiability checks, are usually very expensive in case of MUS enumeration (often NP-complete or harder). Hence, MUS enumeration algorithms often tend to optimize (minimize) the number of performed predicate (unsatisfiability) checks. On the other hand, one can image instances of MSMPs where the predicate checks are very cheap and thus a suitable MSMPs enumeration algorithm should optimize other criteria. Hence, evaluation of the proposed domain agnostic MUS enumeration algorithms for other instances of MSMPs would be an interesting future research direction.

3.1 *Requirements Analysis*

Establishing requirements is an essential stage in all development. In general, the requirements can be described either informally, e.g., using a natural language, or formally via a kind of mathematical logic such as the Linear Temporal Logic (LTL). The formal description enables various model-based techniques, such as model checking. Moreover, we also get the opportunity to check the requirements earlier, even before any system model is built. This so-called requirements sanity checking [Barnat et al., 2012, Bendík, 2017] aims to assure that a given set of requirements is consistent (satisfiable) and that there are no redundancies. If inconsistencies or redundancies are found, it is usually desirable to present them to the user in a minimal fashion, exposing the core problems in the requirements. As redundancy checking can be reduced to inconsistency checking [Barnat et al., 2016], the goal is thus to find either all or at least some minimal inconsistent (unsatisfiable) subsets of requirements. Subsequently, the minimal inconsistent subsets can be used to refine the requirements (see, e.g., our study [Bendík, 2017] on such a methodology).

We illustrate the inconsistency checking on an example adopted from our prior work [Bendík et al., 2016a]. Assume that we are going to build a system that contains one peculiar component that has a very expensive initialization phase and also a very expensive shutdown phase. We constrain the way the component is used by a set of four requirements expressed using the branching temporal logic CTL [Clarke and Emerson, 1981]. In the formulae, we use the atomic propositions q denoting that a *query* has arrived, r denoting that the component is *running*, and m denoting that the system is taken down for *maintenance*. Our first requirement states that whenever a query arrives, the component has to become active eventually; formally $\varphi_1 := \mathbf{AG}(q \rightarrow \mathbf{AF}\, r)$. The second requirement states that once the component is started, it may never be stopped. This may be a reasonable requirement e.g. if the component's initialization is expensive. Formally, the requirement is expressed as $\varphi_2 := \mathbf{AG}(r \rightarrow \mathbf{AG}\, r)$. The third requirement states that the system has to be taken down for maintenance once in a while. This also means that the component has to become inactive at that time. This is formalised as $\varphi_3 := \mathbf{AG}\,\mathbf{AF}\,(m \wedge \neg r)$. Our last requirement states that after the maintenance, the system (including the component we are interested in) has to be restarted, formally $\varphi_4 := \mathbf{AG}(m \rightarrow \mathbf{AF}\,(\neg m \wedge r))$. The situation is illustrated in Figure 3.1. We discover that there is one minimum inconsistent subset of the four requirements, namely $\{\varphi_2, \varphi_3, \varphi_4\}$, and that there are three maximum consistent subsets of the requirements, namely $\{\varphi_1, \varphi_2, \varphi_3\}$, $\{\varphi_1, \varphi_2, \varphi_4\}$, $\{\varphi_1, \varphi_3, \varphi_4\}$. The consistency of the first set $\{\varphi_1, \varphi_2, \varphi_3\}$ might be surprising, as one would suspect the pair of requirements φ_2 and φ_3 to be the source of inconsistency. However, the first three requirements can hold at the same time – in systems where no queries arrive at all.

In general, the MUS enumeration problem naturally subsumes the problem of satisfiability checking, and performing these checks is the most expensive part of the MUS enumeration. In the case of LTL and CTL domains, the satisfiability checking problem is PSPACE-complete [Sistla and Clarke, 1985] and EXPTIME-complete [Emerson and Halpern, 1982], respectively. From the practical point of view, in our recent study [Bendík and Černá, 2018], we were able to deal with constraint sets that contained hundreds of LTL constraints (requirements).

Finally, let us note that our domain agnostic MUS enumeration algorithm, called ReMUS (see Section 4.4), was successfully applied within the European Union's Horizon 2020 project called AMASS[1]. In particular, ReMUS was incorporated into a so-called V&V manager[2]: a tool for validation and verification of a system model and system requirements. The purpose of ReMUS was to enumerate minimal inconsistent (unsatisfiable) subsets of a given inconsistent set of LTL requirements (which were then used to refine the requirements). Our industrial partners evaluated ReMUS on a collection of proprietary industrial benchmarks and compared it with an algorithm tailored for MUS enumeration in the LTL domain called Looney [Barnat et al., 2016]. They found ReMUS to be faster than Looney by several orders of magnitude.

[1] Project AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems), `https://amass-ecsel.eu/` (Accessed: October 2020)

[2] Project AMASS, deliverable D3.6: Prototype for Architecture-Driven Assurance (c), `https://amass-ecsel.eu/content/deliverables` (Accessed: October 2020)

3.2 *Formal Equivalence Checking*

Formal equivalence checking (FEC) [Huang and Cheng, 2012] deals with the following question: Do two design models provide equivalent functionality over all input stimuli?

Performing FEC is often both time and memory expensive even for small models. Therefore, to perform FEC for large models, the compared models are separated into small slices, and instead of checking the complete models for equivalence, the individual slices are checked for equivalence. However, ripping a slice from the complete model may introduce behavior that was not possible in the original model. In particular, some combinations of the input stimuli of the slice may be unrealizable in the complete model. To eliminate such unrealizable behavior, a set of environmental assumptions is constructed and the FEC is performed w.r.t. these assumptions.

One of the contemporary approaches for performing FEC is based on employing SAT solvers. In particular, the two model slices together with the equivalence requirement are encoded into a Boolean formula $\mathcal{R}$, and the environmental assumptions are encoded into a Boolean formula $\mathcal{A}$. The formulae are constructed in a way that their conjunction $\mathcal{R} \wedge \mathcal{A}$ is unsatisfiable if and only if the two slices are functionally equivalent w.r.t. the environmental assumptions. However, to prove that the two slices are indeed functionally equivalent, one must also verify that the environmental assumptions are guaranteed in the complete models. Since verifying the assumptions is very expensive, it is crucial to reduce the number of assumptions that need to be verified.

Cohen et al. [Cohen et al., 2010] noted that $\mathcal{A}$ is usually expressed as a Boolean formula in CNF, i.e., $\mathcal{A}$ can be seen as a set of clauses. Let us say that a subset $\mathcal{A}'$ of $\mathcal{A}$ is *sufficient* if $\mathcal{R} \wedge \mathcal{A}'$ is unsatisfiable. Moreover, $\mathcal{A}'$ is minimal sufficient if none of its proper subsets is sufficient. Cohen et al. proposed to find a minimal sufficient subset $\mathcal{A}'$ of $\mathcal{A}$. Subsequently, only the environmental assumptions that are encoded in $\mathcal{A}'$ need to be verified for holding in the complete model. Even better, multiple minimal sufficient subsets can be found and only the one that encodes the minimum number of assumptions is then processed. The minimal sufficient subsets are a kind of minimal unsatisfiable subsets.

The domain of SAT constraints is the most studied domain in the context of MUS enumeration. The satisfiability problem in this domain is NP-complete [Cook, 1971]. Contemporary MUS enumeration algorithms (e.g., [Bendík et al., 2018b, Liffiton et al., 2016, Narodytska et al., 2018, Bacchus and Katsirelos, 2016]) are able to deal with millions of SAT constraints.

3.3 *Safety Properties Checking*

Another application of MUSes arises in the area of safety properties checking. We are given a model of a system that consists of a set of initial states, a set of all states in the system, a set R of all reachable states, and a set U of unsafe states. The goal is to determine whether the system is safe, i.e., whether $R \cap U = \emptyset$. Due to the combinatorial explosion of the state space (w.r.t. the number of system variables), explicit exploration of the model is practically intractable. Two of the many contemporary approaches for dealing with this problem are *abstraction* and *Bounded Model Checking*.

Using *abstraction* over-approximates the system, i.e., it adds behavior that is not feasible in the original model. This enlarges the set R of all reachable states but also simplifies the description of the model and makes the model checking task easier. If the abstraction is shown to be safe, the original system is necessarily also safe. In the other case, when there is a counter-example to the safety properties in the abstract model, it might be either the case that the counter-example is feasible in the original model or that it is spurious, i.e., caused by the abstraction. The goal is thus to find an abstraction that, on the one hand, extensively simplifies the model and makes the model checking tractable, yet, on the other hand, excludes all spurious counter-examples.

Bounded Model Checking (BMC) deals with the problem of the existence of a counter-example reachable in at most k steps, where k is a fixed number. The problem is encoded into a Boolean formula $\mathcal{F}$ such that $\mathcal{F}$ is unsatisfiable if and only if there is no reachable unsafe state in at most k steps. If $\mathcal{F}$ is satisfiable, we can extract a counter-example from the SAT solver. Otherwise, we can extract a *proof* from the SAT solver that points out why the system is safe up to k steps. The disadvantage of BMC is that if we find no counter-example in at most k steps, we have no guarantee that the system is also safe for larger values of k.

McMillan and Amla [McMillan and Amla, 2003] proposed a technique called *Proof-Based Abstraction Refinement (PBA)* that combines abstraction with BMC. In particular, PBA alternates the two methods, starting with BMC on the original model using a low value of k. If BMC finds a counter-example, the original model is unsafe. In the other case, PBA obtains a proof of $\mathcal{F}$'s unsatisfiability and uses it to build an abstraction of the original model. Such abstraction is guaranteed to be safe up to k steps. PBA checks the abstraction using a standard model checker, and if the abstraction is safe, it is guaranteed that the original model is also safe. In the other case, a counter-example is found and it is checked for being feasible in the original model. If the counter-example is spurious, PBA reruns BMC on the original model with an increased value of k, say k', where k' is the number of steps in which the counter-example was reached in the abstract model.

The less information about the system is used in the proof of $\mathcal{F}$'s unsatisfiability, the more behavior can be abstracted from the original model. This increases the likelihood that the model checking of the resultant abstract model is tractable. It is often the case that $\mathcal{F}$ is expressed as a Boolean formula in CNF, i.e., it is a set of constraints (clauses). In such a case, the minimal proofs of $\mathcal{F}$'s unsatisfiability are the minimal unsatisfiable subsets of $\mathcal{F}$. Therefore, one can enumerate multiple MUSes of $\mathcal{F}$ and use the one with minimum cardinality for building the abstraction. Further optimizations to this procedure were proposed by Nadel [Nadel, 2010].

3.4 *Worst-Case Execution Time Analysis*

Static analysis techniques can be used to compute safe bounds on the worst-case execution time (WCET) of programs. Unfortunately, for large programs, identifying the WCET is often practically intractable within a reasonable time limit. One of the possible approaches to cope with a large program is to run WCET analysis on an abstraction that over-approximates the original program. The abstraction simplifies the description of the system and thus makes the WCET analysis tractable. However, the abstraction might enable paths that are not feasible in the original system, and thus the computed safe bound on the WCET can be very imprecise. To improve the accuracy of the analysis, one has to refine the abstraction. Here, we briefly describe a methodology, called Trickle [Blackham et al., 2014], to automatically detect infeasible paths on compiled binary programs to refine WCET estimates.

The Trickle methodology consists of several steps. In the first step, Trickle transforms the input program into *single static assignment form* (SSA) [Cytron et al., 1991], constructs the control flow graph (CFG) of the program, and computes *conditions* that guard/enable individual transitions/edges in the CFG. All these conditions relate to SSA variables and hence can be expressed as SMT constraints.

In the second step, Trickle employs the *implicit path enumeration technique* [Li et al., 1995] to generate a set of integer linear equations that encode the CFG. Variables of the linear equations represent the execution counts of each basic block, as well as each edge between blocks, in the CFG. These linear equations are used to form an integer linear program (ILP) that captures an abstraction (over-approximation) of the original system. Moreover, an objective function for the ILP is constructed such that its maximum value corresponds to the WCET (worst-case execution time) of the abstraction.

In the next step, Trickle solves the ILP and obtains the WCET. Moreover, based on the values of the variables in the solution of the ILP, Trickle reconstructs the longest path through the control flow graph that corresponds to the WCET.

Subsequently, Trickle checks whether the longest path is feasible in the original system. In particular, Trickle collects all conditions along the path (computed in the first step), i.e. a set C of SMT constraints, and checks C for satisfiability via an SMT solver. If C is unsatisfiable, then the path is infeasible in the original system and thus the abstraction needs to be refined. To find a suitable refinement, Trickle first identifies all minimal unsatisfiable subsets (MUSes) of C. Each such MUS is then used to refine the abstraction of the system (the ILP), eliminating an entire class of infeasible paths including the tested one. Subsequently, Trickle repeats the procedure, i.e., it again solves the (refined) ILP and checks the corresponding path for feasibility until it obtains a path whose corresponding set C of constraints is satisfiable. Such a path is then used as the final upper-bound of the WCET.

Note that in some cases, the MUS enumeration can be practically intractable due to, e.g., a too large number of MUSes or too expensive SMT checks (depending on the underlying SMT theory or simply due to a large number of constraints). In such a case, Trickle is not able to refine the abstraction and thus it terminates and provides at least an over-approximation of the exact WCET (given by the infeasible path). Moreover, note that even in the other case, where the constraints C are satisfiable, the corresponding path might still be infeasible in the original program. Loops, unresolvable memory accesses, non-linear arithmetic, and invariants on the code which are not expressible for the SMT solver, can all create infeasible paths which cannot be found using the Trickle's SMT-based approach.

Finally, let us note that the tractability of a complete MUS enumeration in the SMT domain highly depends on the underlying theory. In Section 4.7, we evaluate several MUS enumeration algorithms on benchmarks from the QF_UF, QF_IDL, QF_RDL, QF_LIA and QF_LRA divisions of the library SMT-LIB[3]. The algorithms can operate on benchmarks that contain millions of SMT constraints, and are able to identify thousands of MUSes within a reasonable time limit.

[3] http://www.smt-lib.org/

4
Domain Agnostic MUS Enumeration

We now gradually describe our two domain agnostic MUS enumeration algorithms: TOME [Bendík et al., 2016b] and ReMUS [Bendík et al., 2018b]. In particular, we first define the notation specific for this chapter (Section 4.1). In Section 4.2, we introduce an MUS enumeration scheme that generalizes several existing MUS enumeration algorithms including TOME and ReMUS. Subsequently, in Sections 4.3 and 4.4, we describe the two specific algorithms. Section 4.5 provides a summary of related work, i.e., other existing domain agnostic MUS enumeration algorithms. In Section 4.6, we describe our domain agnostic MUS enumeration tool that implements several existing MUS enumeration algorithms. Finally, in Section 4.7, we experimentally compare our MUS enumeration algorithms with other existing solutions.

4.1 Notation

Throughout this chapter, we call the elements of C *constraints*, and the monotone predicate $\mathbf{P}$ an *unsatisfiability* predicate. We simply say that a set N is *satisfiable* iff $\mathbf{P}(N) = 0$; otherwise, N is *unsatisfiable*.[1] The $\mathbf{P}_0$-maximal subsets of C are thus *Maximal Satisfiable Subsets* (MSSes) of C, and the $\mathbf{P}_1$-minimal subsets of C are *Minimal Unsatisfiable Subsets* (MUSes) of C:

[1] We assume standard constraint systems where the satisfiability predicate is naturally monotone. Intuitively, an addition of a constraint to an unsatisfiable (also called *over-constrained*) system cannot make it satisfiable. For instance, for the particular case where C is a set of Boolean clauses, we witnessed the monotonicity of the unsatisfiability predicate in Observation 2.2.

Definition 4.1 (MUS). *A subset N of C is a* minimal unsatisfiable subset *(MUS) of C iff N is unsatisfiable and for every $N' \subsetneq N$ it holds that N' is satisfiable. Equivalently, by Observation 2.3, N is an MUS iff for every $c \in N$ the set $N \setminus \{c\}$ is satisfiable.*

Definition 4.2 (MSS). *A subset N of C is a* maximal satisfiable subset *(MSS) of C iff N is satisfiable and for every N', $N \subsetneq N' \subseteq C$, it holds that N' is unsatisfiable. Equivalently, by Observation 2.4, N is an MSS of C iff for every $c \in C \setminus N$ the set $N \cup \{c\}$ is unsatisfiable.*

Critical and conflicting elements for a subset of C (Definitions 2.8 and 2.9) are called *critical and conflicting constraints*, respectively. In particular, a constraint $c \in N$ is critical for an unsatisfiable set N iff $N \setminus \{c\}$ is satisfiable, and a constraint $d \in C \setminus M$ is conflicting for a satisfiable subset M of C iff $M \cup \{d\}$ is unsatisfiable.

```
input : an unsatisfiable set $C$ of constraints
output: all MUSes of $C$
1 Unexplored ← $\mathcal{P}(C)$
2 while there is a u-seed do
3     $S$ ← find a u-seed
4     $crits$ ← collect minable critical constraints for $S$
5     $S_{mus}$ ← shrink($S, crits$)            // black-box single MUS extraction subroutine
6     Unexplored ← Unexplored$\setminus\{T \mid S_{mus} \subseteq T \subseteq C\}$
7     output $S_{mus}$
```

Algorithm 4.1: Seed-Shrink Scheme for MUS enumeration.

As defined in Section 2.3.1, we use the set Unexplored to store all subsets of C that are *unexplored* by an MUS enumeration algorithm, and we follow the rules R1-R4 for manipulation with Unexplored. Furthermore, given an unexplored subset N of C, we say that N is an *s-seed* if N is satisfiable; otherwise, N is a *u-seed*. We also use the terminology of a *correction subset* of C:

Definition 4.3 (MCS). *A subset N of C is a* correction subset *of C iff the set $N\setminus C$ is satisfiable. Moreover, N is a* minimal correction subset *(MCS) of C iff $N\setminus C$ is satisfiable and for every $N' \subsetneq N$ the set $C\setminus N'$ is unsatisfiable.*

Intuitively, MCSes of C represents the minimal subsets of constraints that need to be removed to make C satisfiable. Note that a set N is an MCS of C if and only if the complement $C\setminus N$ of N is an MSS. Consequently, MSSes and MCSes encode the very same information about C's unsatisfiability. Since in some cases it makes sense to talk about satisfiable subsets and in other cases about corrections, we use both the terms in the following.

4.2 Seed-Shrink Scheme

We have proposed [Bendík and Černá, 2020a] a scheme, called *seed-shrink scheme*, that generalizes several existing MUS enumeration algorithms. Especially, the scheme generalizes some of our algorithms that we present in this thesis, and also an algorithm MARCO [Liffiton et al., 2016] which is the major competitor of our domain agnostic algorithms. We call the algorithms that implement the scheme *seed-shrink algorithms*.

The scheme is described in Algorithm 4.1. The computation starts by initializing the set Unexplored to $\mathcal{P}(C)$. Subsequently, the scheme iteratively identifies all MUSes of C. Each iteration starts by finding a u-seed S. Subsequently, minable critical constraints $crits$ for S are collected, and a shrinking procedure (introduced in Section 2.3.3) is used to get an MUS S_{mus} of S. The iteration is concluded by updating the set Unexplored: all supersets of S_{mus} are unsatisfiable since S_{mus} is unsatisfiable, thus they are all removed from Unexplored. The enumeration of MUSes is completed once there is no more u-seed.

All MUSes identified by a seed-shrink algorithm come from shrinking, and any available (even a domain specific) single MUS extraction algorithm can be used to implement the shrinking. Importantly, we assume that the shrinking is implemented in a black-box manner: the input is a u-seed S and a set *crits* of critical constraints for S and the only output is an MUS of S. Therefore, except for identifying the resultant MUS S_{mus}, shrinking has no side effect on the overall computation (e.g., the set `Unexplored` cannot be updated during shrinking).

The scheme does not specify how the u-seeds are found. This can be a very complex process, and the way it is implemented is the only significant difference between the individual seed-shrink algorithms. In general, the process of identifying u-seeds involves checking unexplored subsets for satisfiability and also involves removing satisfiable and/or unsatisfiable subsets (except for MUSes) from the set `Unexplored`. From the efficiency point of view, the main difference between seed-shrink algorithms is in *which* and *how many* subsets the algorithms check for satisfiability while searching for a u-seed. Intuitively, to be efficient, a seed-shrink algorithm should tend to minimize the number of performed satisfiability checks. Also, the u-seeds identified by the algorithm should be relatively small and thus easy to shrink. Moreover, the algorithm should explore $\mathcal{P}(C)$ in a way that allows mining many critical constraints for the u-seeds that are being shrunk.

Finally, let us note that the seed-shrink scheme can be straightforwardly dualized into a *seed-grow* scheme for MSS enumeration. In particular, instead of identifying u-seeds and shrinking them to MUSes, one can identify s-seeds and grow them to MSSes.

4.3 TOME

This section describes our seed-shrink MUS enumeration algorithm called TOME [Bendík et al., 2016b]. The name is an acronym for **T**unable **O**nline **M**US **E**numeration.

4.3.1 *The Basic Workflow*

TOME is an algorithm that enumerates both MUSes and MSSes of the input set of constraints. The algorithm simultaneously implements the seed-shrink and the seed-grow scheme: MUSes are found by shrinking u-seeds and MSSes by growing s-seeds. To perform shrinking and growing efficiently, TOME tends to identify u-seeds and s-seeds that are close (in terms of cardinality) to the resultant MUSes and MSSes and thus should be relatively easy to shrink and grow, respectively. The seeds are found by constructing and searching so-called *unexplored chains*. An *unexplored chain* is a sequence $K = \langle N_1, \dots, N_k \rangle$ such that $N_1 \subseteq N_2 \subseteq \cdots \subseteq N_k$, N_1 is a minimal unexplored subset, N_k is a maximal unexplored subset, and for all $1 < i \leqslant k$ it holds that $|N_i \backslash N_{i-1}| = 1$[2]. The monotonicity of the sat-

[2] Recall that a set N is a minimal unexplored subset iff $N \in$ `Unexplored` and $\forall c \in N$ it holds $N \backslash \{c\} \notin$ `Unexplored`. Dually, a set N is a maximal unexplored subset iff $N \in$ `Unexplored` and $\forall c \in C \backslash N$ it holds $N \cup \{c\} \notin$ `Unexplored`.

```
input : an unsatisfiable set $C$ of constraints
output: all MUSes and all MSSes of $C$
1  Unexplored ← $\mathcal{P}(C)$                                    // a global variable
2  while Unexplored $\neq \emptyset$ do
3      $K_{mss}, K_{mus}$ ← buildAndSearchChain()                 // Algorithm 4.3
4      if $K_{mus}$ is not null then
5          crits ← collect minable critical constraints for $K_{mus}$
6          $U$ ← shrink($K_{mus}$, crits)      // black-box single MUS extraction subroutine
7          output $U$
8          Unexplored ← Unexplored$\setminus\{N | N \supseteq U\}$
9      if $K_{mss}$ is not null then
10         conflicts ← collect minable conflicting constraints for $K_{mss}$
11         $S$ ← grow($K_{mss}$, conflicts)    // black-box single MSS extraction subroutine
12         Unexplored ← Unexplored$\setminus\{N | N \subseteq S\}$
```

Algorithm 4.2: Domain agnostic MUS enumeration algorithm TOME.

isfiability function implies that K has either a *local MUS*, a *local MSS*, or both. They are defined as follows:

1. If all elements of K are satisfiable, then K has no local MUS and N_k is the local MSS of K.

2. If all elements of K are unsatisfiable, then K has no local MSS and N_1 is the local MUS of K.

3. Otherwise, there exists j such that all of $N_1, \ldots, N_j$ are satisfiable and all of $N_{j+1}, \ldots, N_k$ are unsatisfiable. In this case, N_j is the *local MSS* of K and N_{j+1} is the *local MUS* of K.

TOME uses the local MUSes and local MSSes as u-seeds and s-seeds for the shrinking and growing procedures, respectively. The workflow of TOME is shown in Algorithm 4.2. The computation starts by initializing the set Unexplored to $\mathcal{P}(C)$. Each iteration first calls the procedure buildAndSearchChain that builds an unexplored chain and returns the local MSS K_{mss} and the local MUS K_{mus} of the chain. If K_{mus} is defined, the algorithm then collects the minable critical constraints for K_{mus}, shrinks it to an MUS, and updates the set Unexplored. Similarly, if K_{mss} is defined, the algorithm collects the minable conflicting constraints for K_{mss}, grows it to an MSS, and again updates the set Unexplored. The enumeration of all MUSes and all MSSes is completed once there are no more unexplored subsets.

4.3.2 *Building and Searching Unexplored Chains*

The procedure buildAndSearchChain builds an unexplored chain $K = \langle N_1, \ldots, N_k \rangle$, identifies its local MSS K_{mss} and its local MUS K_{mus}, and returns the pair K_{mss}, K_{mus}.

The procedure (shown in Algorithm 4.3) starts by picking the starting node N_1 of the chain, i.e., a minimal unexplored subset. Sub-

```
1 $N_1 \leftarrow$ a minimal unexplored subset of $C$
2 if not checkSat($N_1$) then
3     return null, $N_1$
4 $N_k \leftarrow$ a maximal unexplored subset of $C$ such that $N_1 \subseteq N_k$
5 if checkSat($N_k$) then
6     return $N_k$, null
7 $K \leftarrow$ build an unexplored chain $\langle N_1, \cdots, N_k \rangle$
8 $K_{mss}, K_{mus} \leftarrow$ find the local MSS and the local MUS of $K$ using binary search
9 return $K_{mss}, K_{mus}$
```

Algorithm 4.3: buildAndSearchChain()

sequently, N_1 is checked for satisfiability. If N_1 is unsatisfiable, then every chain that starts with N_1 contains only unsatisfiable nodes. Thus, the procedure terminates and returns the pair (null, N_1). Otherwise, the procedure identifies the node N_k: it picks a maximal unexplored subset N_k such that $N_1 \subseteq N_k$, and checks N_k for satisfiability.[3] If N_k is satisfiable, the procedure returns the pair (N_k, null). Finally, if N_1 is satisfiable and N_k is unsatisfiable, the procedure builds a chain $K = \langle N_1, N_2, \ldots, N_{k-1}, N_k \rangle$ and finds and returns its local MSS K_{mss} and its local MUS K_{mus}; the details are described below.

[3] The details on how we actually obtain the minimal, maximal, and other specific unexplored subsets from `Unexplored` were described in Section 2.3.2.

Recall that the chain K can be divided in two parts: $\langle N_1, \ldots, N_j \rangle$ and $\langle N_{j+1}, \ldots, N_k \rangle$, where $N_1, \ldots, N_j$ are satisfiable and $N_{j+1}, \ldots, N_k$ are unsatisfiable. Therefore, we find the local MSS $N_{mss} = N_j$ and the local MUS $N_{mus} = N_{j+1}$ via a binary search procedure while performing only $\mathcal{O}(\log_2 |K|)$ satisfiability checks.

As for building the chain, we obtain the intermediate nodes N_2, ..., N_{k-1} of K by adding one by one the constraints from $N_k \setminus N_1$ to N_1. The constrains are added in the ascending order imposed by the numbering of the constraints, i.e., $c_1 < c_2 < \cdots < c_n$. For example, given $N_1 = \{c_1, c_5\}$ and $N_k = \{c_1, c_2, c_4, c_5, c_6\}$, we build the chain $\langle \{c_1, c_5\}, \{c_1, c_2, c_5\}, \{c_1, c_2, c_4, c_5\}, \{c_1, c_2, c_4, c_5, c_6\} \rangle$. Such an ordering has a slight advantage over other orderings; for each position on the chain, we can easily compute the set of constraints that belongs to that position. Thus, we do not need to build the chain explicitly. Instead, we compute the sets of constraints that are checked for satisfiability on the fly during the binary search.

Note that the order of the intermediate nodes of the chain highly affects the position of the local MUS and the local MSS on the chain, and thus also the size of the local MUS and local MSS. For example, assume that C is a set of Boolean clauses, $N_1 = \{c_1 = a_1\}$ and $N_k = \{c_1 = a_1, c_2 = a_2, c_3 = a_3, c_4 = a_4, c_5 = \neg a_1\}$. N_1 is satisfiable whereas N_k is unsatisfiable due to the presence of c_1 and c_5 that contradict each other. The position of the local MUS depends on the position of c_5 on the chain (c_1 is fixed to be presented in N_1). Since the size of the u-seed (local MUS) and s-seed (local MSS) highly influences the complexity of shrinking and growing, respectively, one might tend to minimize the size of either local MUSes or local MSSes (in dependence on the relative complexity of shrinking and growing in a particular constraint domain). However, all the intermediate nodes are unexplored and thus the *most suitable* ordering cannot be determined in advance. Yet, there is a space for some domain specific heuristics that at least tend to minimize the size (location) of the local MUSes/MSSes. For example, in the case of Boolean CNF domain, unsatisfiability emerges from a presence of opposite literals (e.g. a and $\neg a$), thus, to minimize the size of the local MUS, one might tend to push clauses with many opposite literals to the beginning of the chain. A study of such heuristics is one of the possible directions for our future work.

4.3.3 *Example Execution of* TOME

Let us now demonstrate the execution of TOME on the set C of four Boolean clauses that we used in the previous examples: $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$. Figure 4.1 shows the values of the control variables of TOME in the individual iterations of the algorithm and also illustrates the current exploration state of $\mathcal{P}(C)$ (i.e., the set `Unexplored`). Moreover, we use the blue color to highlight unexplored chains that are processed, and we also highlight the starting and ending nodes of the chains. We encode subsets of C as bit-vectors. For example, the subset $\{c_1, c_3, c_4\}$ is written as 1011.

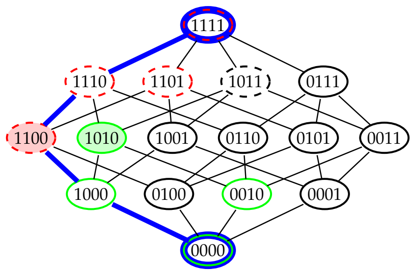
4.3.4 *Optimizations*

Let us now describe various optimizations to the base algorithm. First, recall that every maximal unexplored subset that is satisfiable is an MSS, and dually every minimal unexplored subset that is unsatisfiable is an MUS (Observations 2.12 and 2.13). Thus, in TOME, if a whole unexplored chain $K = \langle N_1, \dots, N_k \rangle$ is unsatisfiable, then its local MUS $K_{mus} = N_1$ is necessarily a (global) MUS. Dually, if the whole K is satisfiable, then its local MSS $K_{mss} = N_k$ is a (global) MSS. Thus, in these situations, we can skip the shrinking or growing and immediately output the MUS or MSS, respectively.

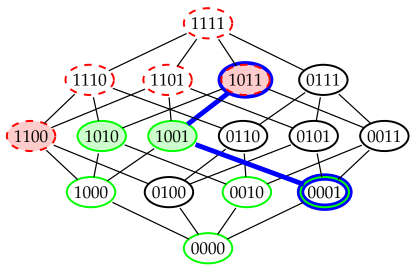
Another slight improvement can be made in the case where both the local MUS K_{mus} and the local MSS K_{mss} are found on an unexplored chain. Since $K_{mus} = K_{mss} \cup \{c\}$ for $c \in K_{mus}$, the constraint c is critical for K_{mus} and conflicting for K_{mss}. Thus, we might add it to the set of critical and conflicting constraints when shrinking K_{mus} and growing K_{mss}, respectively.

Other improvements can be made if we are interested only in the MUS enumeration or only in the MSS enumeration. First, when we identify an MUS M, then from the definition of an MUS, we know that all subsets of M are satisfiable. Thus, when we are updating the set `Unexplored` in line 8 in Algorithm 4.2, we can remove not just the supersets of M but also its subsets. Note that by doing so, we might exclude an MSS from the further computation. Therefore, we can do this only if we do not require the algorithm to identify all MSSes.

I. iteration
– $N_1 = 0000$, found to be satisfiable
– $N_k = 1111$, found to be unsatisfiable
– unex. chain: $K = \langle 0000, 1000, 1100, 1110, 1111 \rangle$
– local MUS $K_{mus} = 1100$, *crits* $= \emptyset$,
shrunk to 1100
– local MSS $K_{mss} = 1000$, *conflicts* $= \emptyset$,
grown to 1010



II. iteration
– $N_1 = 0001$, found to be satisfiable
– $N_k = 1011$, found to be unsatisfiable
– unex. chain: $K = \langle 0001, 1001, 1011 \rangle$
– local MUS $K_{mus} = 1011$, *crits* $= \{c_4\}$,
shrunk to 1011
– local MSS $K_{mss} = 1001$, *conflicts* $= \{c_2\}$,
grown to 1001



III. iteration
– $N_1 = 0011$, found to be satisfiable
– $N_k = 0111$, found to be satisfiable
– unex. chain: not built
– local MUS undefined
– local MSS $K_{mss} = 0111$, *conflicts* $= \{c_1\}$,
grown to 0111

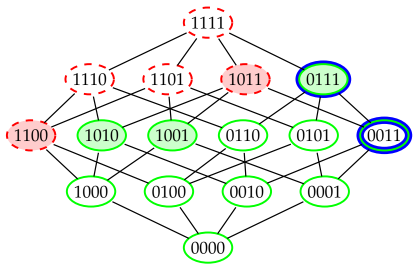


Figure 4.1: An example execution of TOME

Dually, if we are not interested in identifying all MUSes, we can also remove all supersets of all MSSes that are found.

Moreover, if we are interested just in an MUS enumeration, then we can omit growing local MSSes because the growing can be very time demanding and thus slow down the MUS enumeration. Yet, on the other hand, note that it is the explored satisfiable subsets that allow us to collect minable critical constraints. Therefore, identification of satisfiable subsets might significantly speed up the shrinking procedure and consequently speed up the overall MUS enumeration. The choice of whether to grow local MSSes or not thus depends on the particular constraint domain where TOME is applied and on the relative price of growing and the gain given by collecting critical constraints. Again, we can dually apply the same if we are interested only in MSS enumeration.

Finally, there are several domain specific optimizations that we want to discuss since every domain agnostic algorithm (including TOME) is eventually applied in a particular constraint domain. In some constraint domains, such as SAT or SMT, the contemporary satisfiability solvers are often able to provide an *unsat core* when the input set N of constraints is unsatisfiable and a *model* (satisfying variable assignment) when N is satisfiable. The unsat core is often small, yet not necessarily minimal, unsatisfiable subset of N. In TOME, we can use it to reduce the size of a local MUS before we shrink it, i.e., instead of shrinking the local MUS we shrink an unsat core of the local MUS. As for the models, it is often the case that a model π of N satisfies not just all the constraints contained in N but also some constraints in $C \setminus N$. In particular, by the *model extension* of N w.r.t. π we denote the set $\{c | c \in C \textit{ and } \pi \textit{ satisfies } c\}$.[4] In TOME, we can use model extensions to enlarge local MSSes before we grow them, i.e., instead of growing local MSSes, we grow their model extensions.

[4] Recall that in Definitions 2.3 and 2.4, we have properly defined the concepts of an unsat core and a model extension for the SAT domain, i.e., the case when C contains Boolean clauses.

4.4 ReMUS

In this section, we describe our another seed-shrink domain agnostic MUS enumeration algorithm called ReMUS [Bendík et al., 2018b]. The name of the algorithm is an acronym for **R**ecursive **e**numeration of **M**inimal **U**nsatisfiable **S**ubsets.

4.4.1 Basic Idea

Let N be a u-seed and *crits* a set of constraints that are minable critical for N. Recall that, in general, the larger the difference $|N| - |crits|$ is, the harder it is to shrink N. Therefore, an efficient seed-shrink algorithm should search for u-seeds that are small and for which there are many minable critical constraints. To achieve this, ReMUS searches for u-seeds in a so-called *search space*. A *search space* is an unsatisfiable subset of C. Intuitively, the smaller (in terms of cardinality) a search space is, the smaller u-seeds can be found in the search space. ReMUS tries to identify, and then work, in a search space that is relatively small and also contains many minable critical constraints for u-seeds found within.

Initially, the search space is the whole C; during the computation, we recursively refine the search space. We process every search space S iteratively. In each iteration, we pick a maximal unexplored subset S_{max} of S and check it for satisfiability. Based on the result of the satisfiability query, one of the following is performed.

If S_{max} is unsatisfiable, we collect the minable critical constraints for S_{max}, shrink it to an MUS S_{mus}, and mark the MUS as explored. Subsequently, we choose a search space T such that $S_{mus} \subseteq T \subseteq S_{max}$, and recursively process T. Note that any u-seed found within T is smaller than the previous u-seed S_{max} since $T \subseteq S_{max}$. Therefore, the deeper is the recursion, the smaller are the identified u-seeds and, in general, the easier it is to perform the shrinks. We illustrate the search-space reduction in Figure 4.2. Once the recursive call terminates, we continue with the next iteration over the search space S.

If S_{max} is satisfiable, we remove S_{max} together with all its subsets from `Unexplored`. Recall that every maximal unexplored subset that is satisfiable is an MSS (Observation 2.12), thus S_{max} is an MSS of S. Also, since S_{max} is an MSS of S, then every $c \in S \setminus S_{max}$ is critical for $S_{max} \cup \{c\}$. Furthermore, since S_{max} was removed from `Unexplored`, c

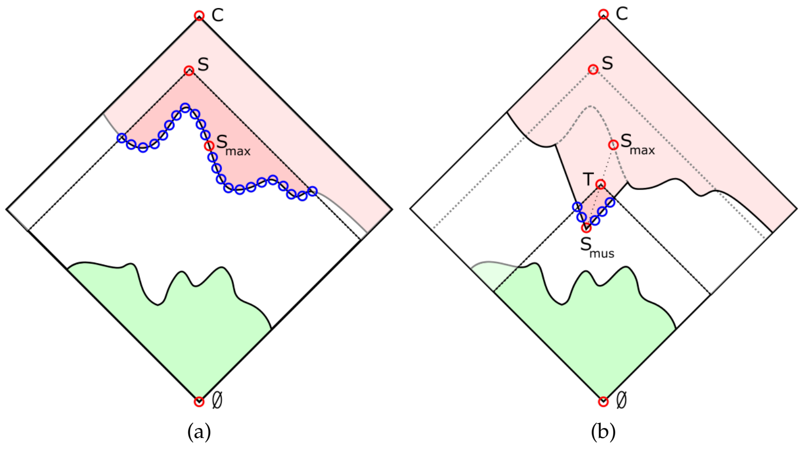


Figure 4.2: Illustration of ReMUS's search space reduction. Both figures show a division of $\mathcal{P}(C)$ into explored satisfiable subsets (green bottom part), unexplored subsets (white middle part), and explored unsatisfiable subsets (top red part). Figure (a) illustrates a search space S, maximal unexplored subsets of S (the blue circles), and a maximal unexplored subset S_{max} of S. In this example, S_{max} is unsatisfiable, thus ReMUS shrinks S_{max} into an MUS S_{mus} and then chooses T, $S_{mus} \subseteq T \subseteq S_{max}$, as a next search space. This is illustrated in Figure (b). We also illustrate maximal unexplored subsets (blue circles) of T in Figure (b).

is also *minable* critical for every u-seed contained in $S_{max} \cup \{c\}$. Based on this observation, we recursively process the search space $S_{max} \cup \{c\}$ for every $c \in S \backslash S_{max}$. Note that these recursive calls also support the tendency to identify small u-seeds since $S_{max} \cup \{c\} \subseteq S$. Once all the recursive calls terminate, we continue with the next iteration over S. The processing of a search space S terminates once $\mathcal{P}(S) \cap$ `Unexplored` $= \emptyset$.

4.4.2 *Complete Description*

We provide the complete description of ReMUS in Algorithm 4.4. The computation starts with the procedure init that initializes all subsets of C to be unexplored, i.e. `Unexplored` $= \mathcal{P}(C)$, and then calls the procedure processSearchSpace(S). The procedure takes as an input a search space S and processes it as described in the previous section. Initially, processSearchSpace is called on the whole C. Bellow, we provide more details on the recursive calls of processSearchSpace.

There are two kinds of recursive calls. The one that is based on a satisfiable maximal unexplored subset S_{max} is performed to move ReMUS into a search space with more minable critical constraints. The other kind of recursion that is based on an unsatisfiable maximal unexplored subset S_{max} moves ReMUS into a smaller search space, i.e., to a search space with smaller u-seeds. Naturally, there is a trade-off between the reduction of the cardinality of the search space and the number of u-seeds that are presented in the search space. The best trade-off differs for particular constraint domains and for particular types of benchmarks. If a benchmark contains a lot of small MUSes, then even a large reduction of the search space might still preserve many u-seeds in the search space. On the other hand, if a benchmark contains only a few MUSes, a fast reduction of the search space soon eliminates all available u-seeds (and yet might keep a lot of s-seeds in the search space). Therefore, in ReMUS (Algorithm 4.4, lines 13 and 14), we do not fix the rate of the search space reduction. We choose a search space T such that $|T| = rf \cdot |S_{max}|$, where rf is a user defined parameter value ranging from 0 to 1. The closer is the value of rf to zero, the more reduced is the cardinality of the search

```
1  Function init(C):
       input : an unsatisfiable set $C$ of constraints
       output: all MUSes of $C$
2      Unexplored ← $\mathcal{P}(C)$                                   // a global variable
3      processSearchSpace($C$)
1  Function processSearchSpace(S):
2      while Unexplored $\cap \mathcal{P}(S) \neq \emptyset$ do
3          $S_{max}$ ← a maximal unexplored subset of $S$
4          if checkSat($S_{max}$) then
5              Unexplored ← Unexplored$\setminus\{N|N \subseteq S_{max}\}$
6              for each $c \in S \setminus S_{max}$ do
7                  processSearchSpace($S_{max} \cup \{c\}$)
8          else
9              $crits$ ← collect minable critical constraints for $S_{max}$
10             $S_{mus}$ ← shrink($S_{max}, crits$)         // black-box single MUS extraction subroutine
11             output $S_{mus}$
12             Unexplored ← Unexplored$\setminus\{N|N \supseteq S_{mus}$ or $N \subseteq S_{mus}\}$
13             if $|S_{mus}| < rf \cdot |S_{max}|$ then
14                 $T$ ← subset such that $S_{mus} \subset T \subset S_{max}$, $|T| = rf \cdot |S_{max}|$
15                 processSearchSpace($T$)
```

Algorithm 4.4: Domain agnostic MUS enumeration algorithm ReMUS.

space in each of the recursive calls. In our experiments (Section 4.7), we set rf to 0.9. We build T by adding the corresponding number of constraints from S_{max} to S_{mus}. The constraints are chosen in the ascending order imposed by the numbering of the constraints, i.e., $c_1 < c_2 < \cdots < c_n$. Note that it might happen that $S_{mus} \geqslant rf \cdot |S_{max}|$. In such a case, we avoid the recursive call altogether.

The set Unexplored is shared among the individual recursive calls. In particular, if ReMUS marks an MUS S_{mus} as explored then all of its supersets w.r.t. the whole $\mathcal{P}(C)$ become explored. Also, let us note that a maximal unexplored subset S_{max} of a search space S, $S \subsetneq C$, might not be a maximal unexplored subset of C.

4.4.3 *Optimizations*

Let us now discuss several optimizations of ReMUS. First, note that we can get into a search space S that contains only a few MUSes. However, to backtrack from S, we first need to explore the whole $\mathcal{P}(S)$, and this might require us to check a large number of maximal unexplored subsets for satisfiability. Consequently, we can be stuck in S for a long time without outputting any MUSes. To prevent this behavior, we detect that S contains a low number of MUSes and pre-emptively backtrack from the recursion. Currently, we use a simple heuristic to detect such a situation. In each local search space S, $S \subsetneq C$, we count the number k of subsequently performed satisfiability checks with the *satisfiable* result. Each identified u-seed resets k to 0. If k exceeds a user-predefined value, we backtrack from S. In our experimental evaluation (Section 4.7), we set k to 10.

Other optimizations that we propose are purely domain specific. Similarly to the case of TOME (Section 4.3.4), we can exploit unsat cores and model extensions in the constraint domains where models and unsat cores are provided by the satisfiability solvers. In particular, when a maximal unexplored subset S_{max} is found to be unsatisfiable, we obtain an unsat core of S_{max}, collect minable critical constraints for the unsat core, and shrink the unsat core instead of shrinking the whole S_{max}. In the other case, where S_{max} is satisfiable, we obtain a model π for S_{max} and compute the model extension $E = \{c | c \in C \textit{ and } \pi \textit{ satisfies } c\}$. Subsequently, we mark E and all of its subsets as explored. Note that E is computed w.r.t. the whole C, i.e., E can outreach the local search space S, and thus $|E|$ can be much larger than $|S_{max}|$. Also, note that w.r.t. $|E| - |S_{max}|$, the number $2^{|E|}$ of subsets of E is exponentially larger than the number $2^{|S_{max}|}$ of subsets of S_{max}. Thus, the use of model extensions significantly speeds up the exploration of $\mathcal{P}(C)$.

4.5 *Related Work*

The early domain agnostic MUS enumeration algorithms were based on the explicit examination of every subset of the unsatisfiable constraint system. As far as we know, the MUS enumeration was pioneered by Hou [Hou, 1994] in the field of diagnosis. Hou's algorithm checks every subset for satisfiability, starting with the whole set of constraints and exploring its power-set in a tree-like structure. Also, some pruning rules that allow skipping irrelevant branches are presented. This approach was revisited and further improved in [Han and Lee, 1999] and [de la Banda et al., 2003]. Another approach using step-by-step power-set exploration was proposed in [Barnat et al., 2016]. The authors of [Barnat et al., 2016] focus on constraints expressed using LTL formulas; however, their algorithm can be used for any kind of constraints. Explicit exploration of the power-set is the bottleneck of all of the above-mentioned algorithms as the size of the power-set is exponential to the number of constraints in the system.

Later algorithms were based on different kinds of a symbolic exploration of the power-set. All the algorithms somehow represent the set `Unexplored` of all unexplored subsets (to be able to exclude already explored MUSes from the computation). Based on the representation of `Unexplored`, we can divide the algorithms into two groups. One group exploits the minimal hitting set duality between MUSes and MCSes (Observation 2.6). The other group maintains the formula $map^{+} \wedge map^{-}$ (the same as we do, first proposed by Liffiton et al. [Liffiton et al., 2016] and described in Section 2.3.2).

```
input : an unsatisfiable set $C$ of constraints
output: all MUSes of $C$
1  $kMUSes \leftarrow kMCSes \leftarrow S \leftarrow \emptyset$
2  repeat
3      $kMCSes \leftarrow C \setminus \mathsf{grow}(S)$
4      $\mathcal{N} \leftarrow$ compute all minimal hitting sets of $kMCSes$
5      $S \leftarrow \emptyset$
6      for $K \in \mathcal{N} \setminus kMUSes$ do
7          if $\mathsf{checkSat}(K)$ then
8              $S \leftarrow K$
9              break
10         else
11             $kMUSes \leftarrow kMUSes \cup \{K\}$
12             output $K$
13 until $S = \emptyset$
```

Algorithm 4.5: DAA.

4.5.1 *Minimal Hitting Set Based Approaches*

Based on our knowledge, the first domain agnostic algorithm that explores the power-set in a symbolic way, and thus can handle input sets with even millions of constraints, was proposed by Bailey and Stuckey [Bailey and Stuckey, 2005] and it is called DAA. Further improvements to DAA were done by Stern et al. [Stern et al., 2012] and presented as an algorithm called PDDS. The two algorithms during their computation identify both MUSes and MCSes of the input constraint set C and store all the already identified MUSes and MCSes in two sets, *kMUSes* and *kMCSes*, respectively. To identify individual MUSes, DAA and PDDS exploit the minimal hitting set duality between MUSes and MCSes. Recall that if a set *kMCSes* is the set of all MCSes of the input formula C, then every minimal hitting set of *kMCSes* is an MUS. If it is the case that *kMCSes* is only a subset of all MCSes of C, then every minimal hitting set of *kMCSes* is either satisfiable or it is an MUS [Bailey and Stuckey, 2005]. Based on this observation, DAA repeatedly computes a minimal hitting set N of the already known MCSes *kMCSes* and checks it for satisfiability. If N is unsatisfiable, it is guaranteed to be an MUS. In the other case, when N is satisfiable, DAA grows N to an MSS and adds the complement of the MSS (i.e., an MCS) to the set *kMCSes*. PDDS is based on a dual approach: it computes minimal hitting sets of the set *kMUSes* of known MUSes, marks the satisfiable hitting sets as MSSes (resp. their complements as MCSes), and shrinks the unsatisfiable hitting sets to MUSes. The exact behavior of the two algorithms is the following.

DAA is described in Algorithm 4.5. The computation starts by initializing the sets *kMUSes* and *kMCSes* to empty sets, and by creating variable S that will be used to store an s-seed for the growing procedure. Initially, $S = \emptyset$ since this is the only subset of C that is

```
input : an unsatisfiable set $C$ of constraints
input : a set $kMUSes$ of MUSes of $C$
input : a set $kMCSes$ of MCSes of $C$
output: all MUSes of $C$
1 $N \leftarrow$ compute a minimal hitting set $N$ of $kMUSes$ such that $C \setminus N$ is unexplored
2 while $N$ is not null do
3     if checkSat($C \setminus N$) then
4         $kMCSes \leftarrow kMCSes \cup \{N\}$
5     else
6         $M \leftarrow$ shrink($C \setminus N$)
7         output $M$
8         $kMUSes \leftarrow kMUSes \cup \{M\}$
9     $N \leftarrow$ compute a minimal hitting set $N$ of $kMUSes$ such that $C \setminus N$ is unexplored
```

Algorithm 4.6: PDDS.

known to be satisfiable. Subsequently, DAA proceeds to two nested loops that form the core part of the algorithm. The outer loop starts by growing the s-seed S into an MSS M of C and adding the complementary MCS $C \setminus M$ to *kMCSes*. Then, DAA computes the set $\mathcal{N}$ of all minimal hitting sets of *kMCSes*, and starts iterating over the new possible candidates for MUSes: $\mathcal{N} \setminus kMUSes$. Each candidate $K \in \mathcal{N} \setminus kMUSes$ is checked for satisfiability. If K is unsatisfiable, then it is necessarily an MUS of C and thus is added to *kMUSes*. In the other case, if K is satisfiable, then the inner loop terminates and K becomes the new s-seed for the growing in the next iteration of the outer loop. The computation of the algorithm terminates once the algorithm fails to find a new s-seed; at this moment, it is guaranteed that all MUSes have been detected.

A weak spot of DAA is the computation of minimal hitting sets. In general, there can be up to exponentially many minimal hitting sets of *kMCSes* w.r.t. $|C|$. Since DAA computes in each iteration all the minimal hitting sets, it can easily run out of memory.

PDDS is described in Algorithm 4.6. The algorithm takes as an input (possibly empty) sets *kMUSes* and *kMCSes* of already known MUSes and MCSes, and iteratively identifies the remaining ones. Each iteration of PDDS starts by computing a minimal hitting set N of *kMUSes* such that $C \setminus N$ is unexplored, and checking N's complement $K = C \setminus N$ for satisfiability. If K is satisfiable, then N is necessarily an MCS of C; otherwise, K is shrunk to an MUS of C. After that, the algorithm continues with the next iteration. The computation terminates once there is no unexplored minimal hitting set of the already identified MUSes; at this point, all subsets of C are explored.

The authors of PDDS note that the algorithm works also in the dual way. That is, instead of computing minimal hitting sets of *kMUSes* and shrinking the unsatisfiable ones, one can compute minimal hitting sets of *kMCSes* and grow the satisfiable ones, similarly to DAA. The main difference between PDDS and DAA is the number of computed minimal hitting sets per iteration: PDDS computes only one minimal hitting set, whereas DAA computes all the minimal hitting sets. Consequently, PDDS does not suffer from such memory issues as DAA does.

Note that neither the authors of DAA nor the authors of PDDS use the term *unexplored subsets* in their work. Thus, let us clarify what are the unexplored subsets in the case of the two algorithms. A subset N of C is unexplored (i.e., $N \in$ `Unexplored`) iff 1) there is no MUS $P \in kMUSes$ such that $N \supseteq P$ and 2) no MCS $Q \in kMCSes$ such that $N \subseteq C \setminus Q$ (since $C \setminus Q$ is an MSS). Note that if a set N is a minimal hitting set of $kMCSes$ such that $N \notin kMUSes$, then N is unexplored. In particular, if N is unsatisfiable, then by the minimal hitting set duality N is an MUS, and since $N \notin kMUSes$, it is an unexplored MUS. In the other case, when N is satisfiable, since N is a hitting set of $kMCSes$, there is no $Q \in kMCSes$ such that $N \subseteq C \setminus Q$ (thus N is unexplored satisfiable subset). Furthermore, since N is a *minimal* hitting set of $kMCSes$, it holds that N is a *minimal unexplored subset* (this was already noted in [Liffiton et al., 2016]). Therefore, the sets that DAA checks for satisfiability are in fact minimal unexplored subsets. Dually, PDDS checks for satisfiability maximal unexplored subsets.

4.5.2 MARCO

Liffiton and Malik [Liffiton and Malik, 2013] and Previti and Marques-Silva [Previti and Marques-Silva, 2013] independently developed two nearly identical algorithms: MARCO [Liffiton and Malik, 2013] and eMUS [Previti and Marques-Silva, 2013]. Both the algorithms were later merged and presented under the name MARCO in [Liffiton et al., 2016]. Currently, the algorithm is perhaps the most widely known domain agnostic MUS enumeration solution. Similarly as DAA and PDDS, MARCO also identifies both MUSes and MCSes (MSSes).

Similarly as TOME, MARCO implements both the seed-shrink and seed-grow schemes. It starts by setting `Unexplored` to $\mathcal{P}(C)$ and then proceeds iteratively. Each iteration starts by getting an unexplored subset S and checking S for satisfiability. If S is satisfiable, i.e., an s-seed, then it is grown into an MSS. In the other case where S is unsatisfiable, i.e, a u-seed, S is shrunk into an MUS. In both cases, the set `Unexplored` is appropriately updated. The computation terminates once there is no more unexplored subset.

The authors of MARCO proposed three different strategies for selecting unexplored subsets. The first strategy is to pick an unexplored subset at random, the second strategy is to pick a minimal unexplored subset, and the third strategy is to pick a maximal unexplored subset. Recall that a minimal unexplored subset that is unsatisfiable is an MUS (Observation 2.13). Dually, a maximal unexplored subset that is satisfiable is an MSS (Observation 2.12). Therefore, in the second strategy, the set S is either satisfiable and it is grown to an MSS, or it is guaranteed to be an MUS, and thus the shrink is omitted. Similarly, in the third strategy, the set S is either unsatisfiable and it is shrunk to an MUS, or it is guaranteed to be an MSS, so the grown is omitted. The authors of MARCO proposed the second strategy to be used when the goal is to bias the enumeration for MSSes and the third strategy when the goal is to bias the enumeration for MUSes. At the first glance, one might expect the second strategy to be better for an MUS enumeration since we can skip the shrinking. However, in practice, unsatisfiable subsets of C are naturally more concentrated among the larger subsets of C and, dually, satisfiable subsets are naturally more concentrated among the smaller subsets of C. Since maximal (minimal) unexplored subsets are usually very large (small), it is mostly the case that the maximal (minimal) unexplored subsets are unsatisfiable (satisfiable). Consequently, MARCO usually quickly finds a u-seed among the maximal unexplored subsets and, thus, is able to perform shrinks (and identify MUSes) at a relatively steady rate.

Note that the variant of MARCO that is optimized towards MUS enumeration works in a similar way as PDDS, since maximal unexplored subsets used in MARCO are equivalent to the unexplored complements of minimal hitting sets of *kMUSes* used in PDDS. However, the authors of PDDS do not specify how to obtain the minimal hitting sets; they state that it can be done by any available approach. On the other hand, the authors of MARCO were the first who proposed to use the symbolic representation of unexplored subsets based on the formula $map^+ \wedge map^-$. Their symbolic representation was subsequently adopted by many other MSS or MUS enumeration algorithms including the our ones (please, refer back to Section 2.3.2 for details). Furthermore, authors of MARCO proposed to collect minable critical/conflicting constraints for u-seeds/s-seeds before shrinking/growing them. Also, same as in our algorithms, MARCO employs unsat cores and model extensions to reduce and increase the size of the u-seeds and s-seeds, respectively, in constraint domains where unsat cores and model extensions are provided by a satisfiability solver. For a more detailed comparison of MARCO and PDDS, please refer to [Liffiton et al., 2016].

Besides the above mentioned domain agnostic MUS enumeration algorithms, there have been proposed many MUS enumeration algorithms that are tailored for a particular constraint domain (e.g., [Arif et al., 2015b, Bacchus and Katsirelos, 2015, 2016, Narodytska et al., 2018]). These algorithms extensively exploit specific properties of the particular domain and cannot be used in other domains (we discuss some of these algorithms later in Section 5.3). Also, there have been proposed several algorithms that enumerate MUSes *offline*, i.e., the algorithms either find all MUSes within a given time limit or find no MUS at all. For example, CAMUS [Liffiton and Sakallah, 2008] first identifies all MCSes of C and then uses the minimal hitting set duality between MUSes and MCSes to obtain the MUSes from the MCSes. The problem with CAMUS is that if C contains a large number of MCSes, then no MUS is identified within the time limit. Finally, there have been several studies on domain agnostic algorithms for a single MUS extraction. For instance, see the work by Janota and Marques-Silva [Janota and Marques-Silva, 2016] on the complexity of domain agnostic single MUS extraction.

4.6 MUST: *A Domain Agnostic MUS Enumeration Tool*

In this section, we describe our tool that implements TOME and ReMUS. Before we jump into technical details, let us first remind what is the purpose of developing domain agnostic MUS enumeration algorithms. As opposed to a domain specific algorithm that is highly optimized towards a particular constraint domain, the purpose of a domain agnostic algorithm is mainly to provide a ready-to-use solution for an arbitrary constraint domain where MUSes might eventually find an application. As we have discussed in the previous sections, there have been proposed several domain agnostic MUS enumeration algorithms in the past two decades. Yet, there is almost no available *domain agnostic tool implementation* of the algorithms that would actually serve as a ready-to-use solution for an arbitrary constraint domain. Papers that present existing domain agnostic algorithms usually provide results of an experimental evaluation, however, it is often the case that the implementation is either not publicly available [Barnat et al., 2016, Bailey and Stuckey, 2005], or there is a hard-coded support for a particular constraint domain [Bendík et al., 2018c, Ghassabani et al., 2017b]. The closest to a domain agnostic tool is a tool by Liffiton et al. [Liffiton et al., 2016] where the authors implement their domain agnostic MUS enumeration algorithm MARCO. Their tool currently supports the SAT and the SMT domains and can be relatively easily extended to support also another constraint domains. However, our experimental evaluation [Bendík and Černá, 2018] of contemporary domain agnostic algorithms in various constraint domains has shown that the efficiency of the algorithms (including MARCO) varies a lot in different constraint domains. There is no silver bullet algorithm that would be efficient in all the domains. Thus, to deal with a particular constraint domain, one has to wisely choose from individual algorithms.

To close this gap, we developed an MUS enumeration tool called MUST [Bendík and Černá, 2020a]. The tool implements three domain agnostic algorithms based on the seed-shrink scheme: MARCO [Liffiton et al., 2016], and our TOME and ReMUS. Currently, the tool provides support for 3 constraint domains: SAT, SMT, and LTL. Moreover, due to a modular architecture of the tool, the tool can be easily extended to support another constraint domain: it requires only to implement an API for communication with a satisfiability solver for the constraint domain. Therefore, MUST can be seen as the first *domain agnostic MUS enumeration tool*: the user of the tool can easily adapt and apply the tool in an arbitrary constraint domain. The tool is implemented in C++ and it is available at:

https://github.com/jar-ben/mustool

In the following, we briefly describe the architecture of the tool (Section 4.6.1), discuss how are the currently supported satisfiability solvers implemented and how to add a support for additional constraint domains (Section 4.6.3), and how to install and run the tool (Section 4.6.3). An experimental comparison of the tool with other (even domain specific) MUS enumeration tools is provided in Section 4.7.

4.6.1 Logical Components

The tool consists of six logical components: *SatSolver*, *Explorer*, *Master*, *Algorithms*, *Heuristics*, and *Initializer*.

SatSolver *SatSolver* (declared in *SatSolver.h*) is the only domain specific part of our tool. It provides the functionality for checking sets of constraints for satisfiability, and implements the shrinking procedure. Also, *SatSolver* copes with parsing the input set of constraints (provided by the user) and exporting the identified MUSes in particular domain specific formats. A more detailed description of *SatSolver* is provided in Section 4.6.2.

Explorer *Explorer* (declared in *Explorer.h*) stores the set `Unexplored` of all unexplored subsets and handles related operations including: marking sets as explored, obtaining unexplored subsets, and mining critical and conflicting constraints.

Master *Master* (declared in Master.h) is the coordinator of the whole computation. In particular, it holds an instance of Explorer and an instance of SatSolver and provides wrappers for calling their methods. Moreover, it runs an MUS enumeration algorithm that is specified by the user via a command line argument (see below).

Algorithms The algorithms, MARCO, TOME, and ReMUS are declared in Master(.h) and implemented in marco.cpp, tome.cpp, and remus.cpp, respectively. All calls to SatSolver and Explorer are made via the wrappers defined in Master. This means that any improvement to Explorer and especially to SatSolver (i.e. a more efficient shrinking procedure or satisfiability solver) is immediately reflected by all the algorithms.

Heuristics There are several heuristics that are bound to the wrappers defined in Master, and thus can be exploited by all the three algorithms. For example, in the wrapper for invoking the shrinking procedure, we provide two heuristics for computing critical constraints for the set that is being shrunk. One of the two heuristics uses Explorer to compute critical constraints based on the set `Unexplored`. The other heuristic uses SatSolver to obtain additional critical constraints that cannot be mined from `Unexplored`.

Initializer *Initializer* (implemented in main.cpp) parses the command line arguments provided by the user, and creates, sets-up, and runs the Master.

4.6.2 SatSolver

SatSolver (declared in *SatSolver.h*) is an abstract class stating all the domain specific functionality that needs to be implemented (in a derived class) to support a particular constraint domain in our tool. There are three methods that have to be implemented by every derived class:

- `toString(`N`)` takes as an input a set N, $N \subseteq C$, and returns a textual representation of the constraints contained in N (e.g. in the SMT-LIB 2 format if N is a set of SMT constraints). We use this method to output the identified MUSes.
- `solve(`N`,` *core* = *False*`,` *extension* = *False*`)` takes as an input a subset N of C and returns *True* iff N is satisfiable and *False* otherwise. Moreover, `solve` takes two optional Boolean parameters, *core* and *extension*, with default values set to *False*. If *core* is set to *True* and N is unsatisfiable, `solve` also finds an *unsat core* of N, i.e. an unsatisfiable M such that $M \subseteq N$. Similarly, if *extension* is set to *True* and N is satisfiable, `solve` finds a *model extension* of N, i.e. a satisfiable set M such that $N \subseteq M \subseteq C$. We use the unsat cores in our tool to reduce seeds before shrinking. The extensions are used to further prune the set `Unexplored` when an unexplored subset is found to be satisfiable.
- `constructor(`*filepath*`)`. Every derived class of SatSolver has to implement its constructor. The constructor accepts a path *filepath* to a file that specifies the input set C of constraints in some domain specific format (e.g. SMT-LIB 2 for SMT formulae). We invoke the constructor during the initialisation phase of our tool and its goal is to parse the input set of constraints and internally store the constraints for future manipulations. SatSolver is the only one of the six logical components of our tool that directly works with particular constraints of C. All the other components work just with a bit-vector representation of subsets of C. For example, if $C = \{c_1, c_2, c_3, c_4\}$ is a set of four constraints and $K = \{c_1, c_2\}$, the bitvector representation of K is 1100. Therefore, whenever another component communicates with SatSolver, e.g. invokes the procedure `solve(`N`)`, it passes the bit-vector representation of N to SatSolver and SatSolver converts it to particular constraints.

Besides the above three methods that have to be implemented by every derived class, SatSolver defines and implements a method that can be overridden by a derived class:

- `shrink(`N`,` *crits*`)` performs the shrinking, i.e. it takes an unsatisfiable set N together with a set *crits* of constraints that are critical for N and returns an MUS of N. The default domain agnostic implementation of this method is carried out by Algorithm 2.1 (Section 2.3.3).

Currently, our tool supports three constraint domains via the following four derived classes of SatSolver:

- **MSHandle** (implemented in *MSHandle.cpp*) provides a functionality for the Boolean CNF domain, i.e. the set of constraints is a set of Boolean clauses. The input and output format is the DIMACS CNF format. For shrinking, we integrate two single MUS extraction tools: muser2 [Belov and Marques-Silva, 2012] and mcsmus [Bacchus and Katsirelos, 2015]. Finally, we use miniSAT [Eén and Sörensson, 2003] to implement the method `solve`. Besides checking N for satisfiability, we also use miniSAT to obtain an unsat core or a model extension of N. In particular, an unsat core is directly provided by miniSAT. To get a model extension of N, we obtain a model π of N from miniSAT and collect the set $\{c | c \in C \wedge \pi \models c\}$ of all constraints in C that are satisfied by π.
- **Z3Handle** (implemented in *Z3Handle.cpp*) can process SMT constraints represented in the SMT-LIB2 format. We use z3 [de Moura and Bjørner, 2008] to parse the input and to implement `solve`. Moreover, in the same way as in the case of MSHandle, we obtain unsat cores from z3 and we also obtain models of satisfiables formulas to compute their extensions. The shrinking is implemented using our custom procedure.
- **SpotHandle** (implemented in *SpotHandle.cpp*) supports the LTL domain. We use SPOT [Duret-Lutz et al., 2016] to implement `solve` and the default domain agnostic implementation of `shrink`. In this case, we do not provide support for computing *non-trivial* unsat cores and *non-trivial* extension. Therefore, if an extension or unsat core is required while calling `solve(`N`)`, we simply use N itself (N is a trivial unsat core/extension of N).
- **NuxmvHandle** (implemented in *NuxmvHandle.cpp*) is another alternative for the LTL domain. It uses nuXmv [Cavada et al., 2014] as a satisfiability solver, which is, based on our experience, much more efficient than SPOT. However, nuXmv's license[5] is more restrictive than the SPOT's license and thus not every user of our tool might use it. In this case, we also do not support an extraction of non-trivial unsat cores and extensions.

[5] `https://es-static.fbk.eu/tools/nuxmv/index.php?n=Main.License`

If anyone wants to add support for another constraint domain to our tool, it is enough to implement a derived class of SatSolver. For example, the implementation of SpotHandle takes only 45 lines of code, including several empty lines caused by formatting and lines containing only closing brackets ("}"). Therefore, we claim our tool to be indeed domain agnostic and ready-to-use solution for any constraint domain.

4.6.3 *Installation and Execution of the Tool*

The tool is actively maintained and the up-to-date installation and usage instructions are available at: `https://github.com/jar-ben/mustool`.

Briefly, our tool can be built either in lightweight settings with support only for SAT domain, or with support also for the SMT and/or LTL domains. Whereas in the SAT domain, we use miniSAT that can be built very quickly, the z3 and SPOT solvers that we use in the SMT and LTL domains can take several hours to install. Once you have installed all the solvers you want to use, our tool can be simply built with an invocation of the command "make".

To run our tool in its default settings, execute:

```
./must input_file,
```

where `input_file` specifies the input file of constraints, and it has to have either .cnf, smt2, or .ltl extension. Based on the extension, Master selects and uses an appropriate derived class of SatSolver. To specify an MUS enumeration algorithm to be used, invoke the tool by:

```
./must -a alg input_file,
```

where `alg` can be either `marco`, `tome`, or `remus` (the default one). To see all the available settings, run

```
./must -h.
```

4.7 *Experimental Evaluation*

We now provide an experimental comparison of contemporary domain agnostic MUS enumeration algorithms MARCO, TOME, and ReMUS. The comparison is done in three constraint domains: SAT, SMT, and LTL. Note that we do not include the algorithm DAA in the evaluation, although we have described it in Section 7.2. That is because DAA has been already shown to be very inefficient in several previous papers [Liffiton and Malik, 2013, Liffiton et al., 2016, Bendík and Černá, 2018], and thus we rather devote the available space to the analysis of the other algorithms. Finally, in the SAT domain, we compare the three domain agnostic algorithms also with the two contemporary approaches tailored to that constraint domain: FLINT [Narodytska et al., 2018] and MCSMUS [Bacchus and Katsirelos, 2016].

All experiments were run using a time limit of 3600 seconds and computed on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux 4.19.67-2. The comparison criterion used in our evaluation is the number of identified MUSes within the given time limit. Complete results are available in the online appendix[6].

[6] https://www.fi.muni.cz/~xbendik/phdThesis/

4.7.1 *Implementations*

Both ReMUS and TOME, with a support for all the three constraint domains, are implemented in our MUS enumeration tool. We use miniSAT [Eén and Sörensson, 2003], Z3 [de Moura and Bjørner, 2008], and nuXmv [Cavada et al., 2014], as satisfiability solvers in the SAT, SMT, and LTL domain, respectively. In the SAT domain, we use a single MUS extraction subroutine of the MCSMUS tool [Bacchus and Katsirelos, 2015] for shrinking; in the LTL and SMT domains, we use our custom shrinking procedure. Our tool is publicly available on github:

https://github.com/jar-ben/mustool

In the case of MCSMUS, we used the latest (last commit in May 2019) publicly available implementation by George Katsirelos that is available at:

https://bitbucket.org/gkatsi/mcsmus

FLINT was kindly provided to us by its author, Nina Narodytska. As for MARCO, in the SAT and SMT domains, we used the latest (version 2.0.1) publicly available implementation of MARCO by Mark H. Liffiton:

https://sun.iwu.edu/%7emliffito/marco/

Since the implementation of MARCO by Liffiton does not support the LTL domain, we have reimplemented MARCO in our MUS enumeration tool. The reimplementation is based on the original implementation of MARCO by Liffiton; the core part of the algorithm behaves exactly the same, we have just implemented a satisfiability solver and a shrinking procedure for the LTL domain. In particular, we used the same shrinking procedure and the same satisfiability solver for MARCO as we did for ReMUS and TOME.

4.7.2 Boolean Domain

Benchmarks We used a collection of 291 Boolean CNF benchmarks that were taken from the MUS track of the SAT 2011 Competition[7]. This collection has been used in many papers, e.g., [Narodytska et al., 2018, Liffiton et al., 2016, Bacchus and Katsirelos, 2016, Bendík et al., 2016b, 2018b], that focus on MUS enumeration. The benchmarks range in their size from 70 to 16 million constraints and use from 26 to 4.4 million variables. In the case of 27 benchmarks, all the evaluated algorithms identified all the MUSes within the given time limit. Since the comparison criterion of our evaluation is the number of identified MUSes, the 27 benchmarks are irrelevant for the evaluation (all algorithms found the same number of MUSes for these benchmarks). Therefore, only the remaining 264 benchmarks are the subject of our evaluation.

[7] http://www.cril.univ-artois.fr/SAT11/

Results In Figure 4.3, we provide scatter plots that compare pair-wise individual algorithms. Each point in a scatter plot corresponds to a single benchmark and shows the number of identified MUSes by the two algorithms. The x-coordinate of a point is given by the algorithm that labels the x-axis, and the y-coordinate of a point is determined by the algorithm that labels the y-axis. Intuitively, points below the diagonal favour the algorithm that labels the x-axis and vice versa. Since the plots are in a log-scale, i.e., they cannot show plots with a zero coordinate (zero MUSes), we lifted the points with a zero coordinate to the first coordinate. In other words, points that lay exactly on the axis represent benchmarks where one of the algorithms

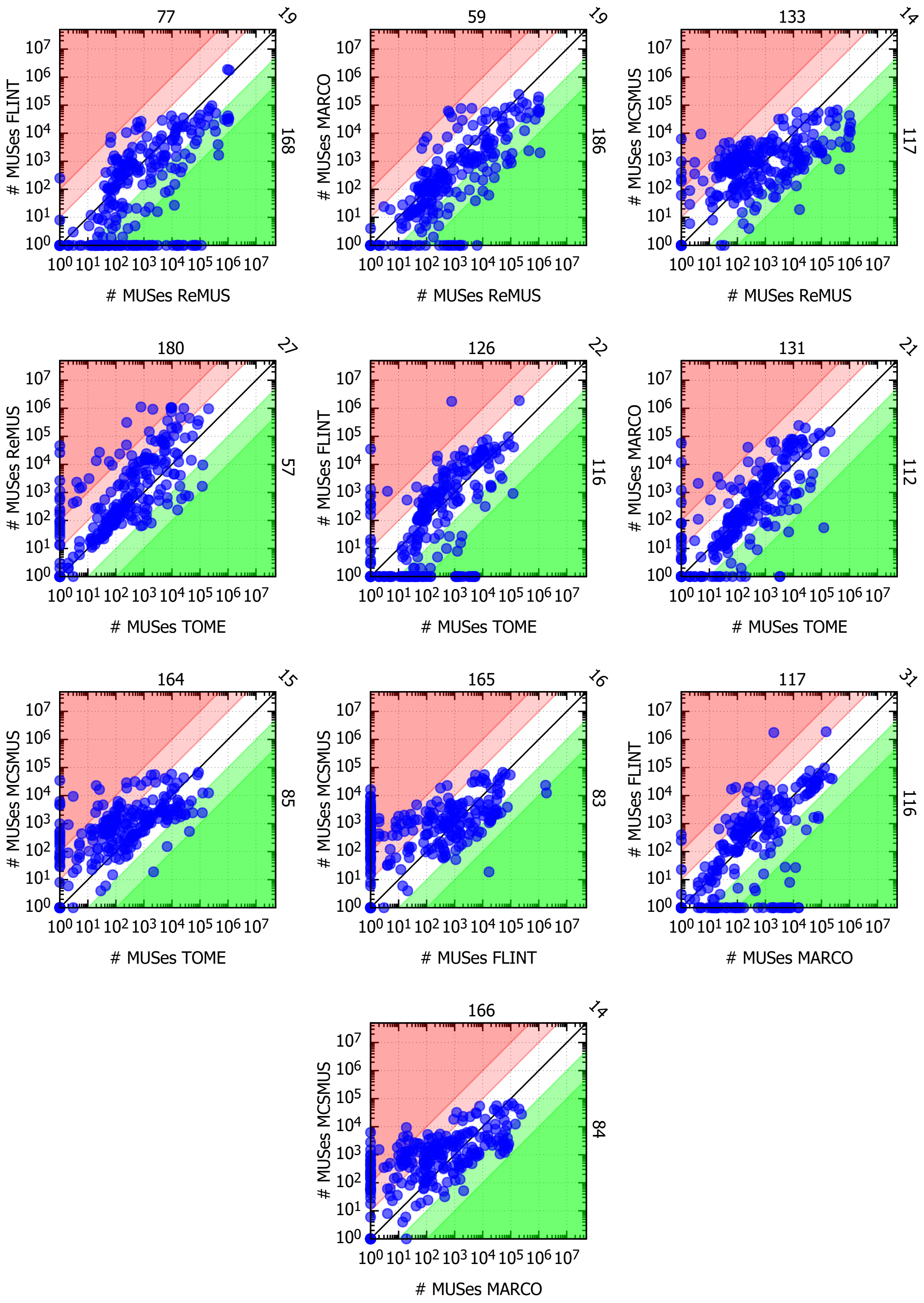


Figure 4.3: Scatter plots comparing the number of produced MUSes in the SAT domain.

found either one or no MUS. Furthermore, we provide three numbers right/above/in the right corner of the plot, that show the number of points below/above/on the diagonal. For instance, ReMUS found more/less/equal number of MUSes as FLINT in case of 168/77/19 benchmarks. Finally, we use green and red colors to highlight individual orders of magnitude (of 10).

Besides the pair-wise comparison of the algorithms, we also provide an overall *ranking* of the algorithms on individual benchmarks. In particular, assume that for a benchmark B both MCSMUS and ReMUS found 100 MUSes, MARCO found 80 MUSes, and both TOME and FLINT found 50 MUSes. In such a case, MCSMUS and ReMUS share the 1st (best) rank for B, MARCO is 3rd, and TOME and FLINT share the 4th position. Table 4.1 shows a summary of this ranking. In particular, for each algorithm, the table shows the average ranking (avg) of the algorithm w.r.t. all benchmarks, and also the number of benchmarks where the algorithm ranks as 1st, 2nd, 3rd, 4th, and 5th, respective.

The best average ranking, 2.23, was achieved by ReMUS, and the second-best, 2.4, was achieved by MCSMUS. These two algorithms also dominate in being ranked as 1st; MCSMUS and ReMUS were the best in the case of 111 and 106 benchmarks, respectively. As for the remaining three algorithms, they all achieved a ranking between 3.14 and 3.28, i.e., there is no big difference between these three algorithms w.r.t the ranking criterion.

	ranked 1st	ranked 2nd	ranked 3rd	ranked 4th	ranked 5th	average ranking
FLINT	45	59	36	62	62	3.14
MARCO	33	58	50	69	54	3.2
MCSMUS	111	39	42	42	30	2.4
ReMUS	106	59	48	33	18	2.23
TOME	29	47	73	52	63	3.28

Table 4.1: Overall rankings of evaluated algorithms in the SAT domain.

To conclude the evaluation in the SAT domain, we state that there is no algorithm that would actually defeat all the other algorithms on a *vast majority* of benchmarks. Although it was usually the case that either ReMUS or MCSMUS performed the best, there are tens of benchmarks where one of the other three algorithms dominated. Clearly, each of the evaluated algorithms is suitable for some kind of benchmarks. We provide more insight into the dependency on the type of benchmarks in Section 4.8.

4.7.3 *SMT Domain*

Benchmarks We conducted the experiments on a collection of 433 benchmarks that were taken from the QF_UF, QF_IDL, QF_RDL, QF_LIA and QF_LRA divisions of the library SMT-LIB[8]. This collection has been, for example, used in the work by Griggio et al. [Cimatti et al., 2011] and in our recent papers [Bendík et al., 2018b, Bendík and

[8] `http://www.smt-lib.org/`

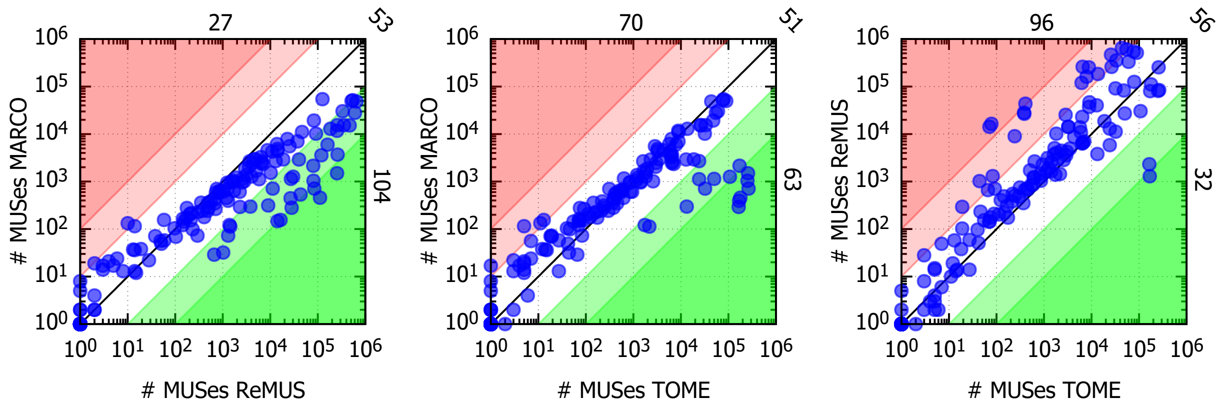


Figure 4.4: Scatter plots comparing the number of produced MUSes in the SMT domain.

Černá, 2018]. The benchmarks range in their size from 2 to 32808 constraints. In the case of 249 benchmarks, all the evaluated algorithms identified all the MUSes. In the case of the remaining 184 benchmarks, no algorithm identified all MUSes. Therefore, we focus here on the 184 benchmarks.

Results Scatter plots comparing pair-wise the evaluated algorithms on individual benchmarks are provided in Figure 4.4. Similarly as in the case of the SAT domain, each scatter plot is labeled with three numbers right/above/in the right corner of the plot that show the number of points below/above/on the diagonal. The plots are in a log-scale and points with a zero coordinate are moved to the first coordinate.

Moreover, in Table 4.2 we provide the overall ranking of the algorithms. ReMUS significantly dominates both its competitors; in the case of 138 benchmarks, ReMUS performed the best, and only in case of 13 benchmarks, it performed the worst. TOME and MARCO are competitive to each other in terms of the ranking criterion as well as in terms of the scatter plot pair-wise comparison.

	ranked 1st	ranked 2nd	ranked 3rd	average ranking
MARCO	75	51	58	1.91
ReMUS	138	33	13	1.32
TOME	77	48	59	1.9

Table 4.2: Overall rankings of evaluated algorithms in the SMT domain.

4.7.4 *LTL Domain*

Benchmarks Although LTL has been widely used, especially in connection with LTL model checking, there are no publicly available LTL MUS benchmarks. Namely, LTL model checking benchmarks (e.g., the collection BEEM [Pelánek, 2007]), cannot be used since these LTL formulas are consistent w.r.t. to each other; their inconsistency emerges only after adding a model of a system. Therefore, we have followed the approach from [Barnat et al., 2016] and generated custom MUS benchmarks using the tool `randltl` from the SPOT library [Duret-Lutz et al., 2016]. To obtain as realistic bench-

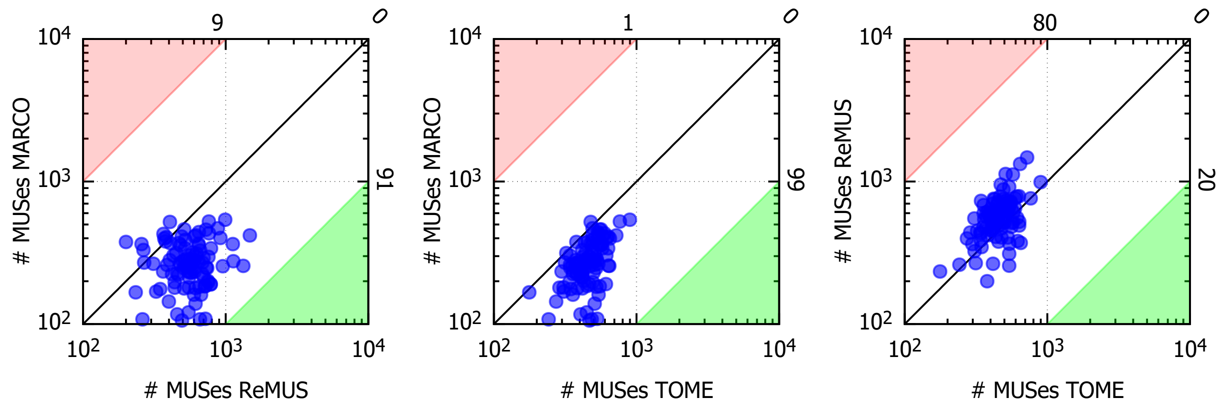


Figure 4.5: Scatter plots comparing the number of produced MUSes in the LTL domain.

marks as possible, we reflected the statistics from [Dwyer et al., 1998] about the most common industrial LTL formulas. We generated 100 benchmarks that use up to 15 atomic propositions (variables) and range in their size from 145 to 238 formulas. Note that compared to the benchmarks used in the SAT and SMT domains, the generated benchmarks are relatively small. This is caused by the complexity of the satisfiability problem in the LTL domain. Larger benchmarks would be intractable to deal with.

Results We compare the algorithms via scatter plots in Figure 4.5. It was the case that all evaluated algorithms on every benchmark identified at least 100 MUSes and less than 10000 MUSes, therefore we show only this area in the plots. Moreover, as in the case of SAT and SMT domains, we show the number of points below/above/on the diagonal using the numbers right/above/one the diagonal.

ReMUS conclusively outperformed MARCO on almost all of the instances; there were only 9 benchmarks where MARCO performed better than ReMUS. As for TOME and MARCO, there is only 1 instance where MARCO performed better than TOME. Finally, ReMUS beats TOME on a majority of the benchmarks, yet note that there are several benchmarks where TOME performed significantly better than ReMUS. As usually, we complement the pair-wise comparison of the algorithms with Table 4.3 that shows the overall ranking of the algorithms.

	ranked 1st	ranked 2nd	ranked 3rd	average ranking
MARCO	1	8	91	2.9
ReMUS	80	11	9	1.29
TOME	19	81	0	1.81

Table 4.3: Overall rankings of evaluated algorithms in the LTL domain.

Finally, let us note that we have already compared the three algorithms before [Bendík and Černá, 2018] using the same set of benchmarks, and this time, ReMUS performs better than in the previous evaluation. This is caused by introducing the pre-emptive backtracking as described in Section 4.4.3.

4.8 Discussion About Results And Recommendations

Since the main purpose of this chapter is to discuss domain agnostic algorithms, we mainly focus on the comparison of MARCO, TOME, and ReMUS. All the three algorithms are based on the seed-shrink scheme and thus, from the high-level view, work quite similarly. Yet, we have seen that whereas ReMUS performed very well in all the three constraint domains, the performance of MARCO and TOME was not so stable across the individual domains. Let us thus discuss what constraint domains are suitable for the individual algorithms.

MARCO searches for a u-seed among the maximal unexplored subsets. Since the unsatisfiable subsets are naturally more concentrated among the larger subsets, this kind of search usually allows MARCO to find a u-seed by checking only few unexplored subsets for satisfiability. However, maximal unexplored subsets are mostly very large and thus generally hard to shrink. Yet, MARCO performed quite well in the SAT and SMT domains. The reason is that the SAT and SMT domains are very specific in the ability of satisfiability solvers to extract unsat cores of the input formula. In particular, the extracted cores are often very close (in terms of cardinality) to the MUSes of the formula. Moreover, the extraction usually comes with almost no overhead. Consequently, in these domains, the size of a u-seed for the shrinking procedure is not an important factor, since the u-seed can be cheaply reduced via its unsat core. What mainly comes into play is thus the number of satisfiability checks that are performed in order to find the u-seed, and MARCO is very frugal in this criterion.

On the contrary, in the LTL domain, unsat core extraction is not a common feature of the contemporary satisfiability solvers. Also, shrinking in the LTL domain has not been yet studied so extensively as in the SAT [Belov and Marques-Silva, 2012, Belov et al., 2014, Nadel et al., 2014] and the SMT [Guthmann et al., 2016] domains, and the size of the u-seeds indeed highly affects the complexity of the shrinking. Consequently, MARCO is not so efficient in this domain.

TOME tends to identify small u-seeds for shrinking by searching unexplored chains with binary search, and it takes only $\mathcal{O}(\log_2 |C|)$ satisfiability checks to process an unexplored chain. Based on our experience, the u-seeds found by TOME are indeed usually notably smaller than the u-seeds identified by MARCO. That is why TOME significantly outperforms MARCO in the LTL domain. We recommend using TOME mainly in constraint domains where satisfiability solvers do not enjoy the ability to extract small unsat cores efficiently.

Also, TOME is the only algorithm in our evaluation that can identify an MUS via a single check for satisfiability: by checking minimal unexplored subsets. Recall that minimal unexplored subsets correspond to minimal hitting sets of the already identifies MCSes. Moreover, the larger percentage of MCSes is already identified, the higher is the chance that a minimal unexplored subset is an MUS. Therefore, TOME is very efficient in the case of benchmarks that contain a relatively small number of MCSes. In such benchmarks, TOME outperforms all the other evaluated algorithms.

ReMUS also tends to identify small u-seeds; however, it uses a different approach then TOME. Based on our experience, ReMUS usually finds smaller u-seeds than TOME, and it also performs notably fewer satisfiability checks than TOME to find the u-seeds. Moreover, it explores the power-set in a way that allows to mine many critical constraints. ReMUS significantly outperforms TOME and MARCO on a majority of the benchmarks in all the three constraint domains, and thus it is the clear winner of our comparison. Moreover, in the SAT domain, ReMUS outperforms the domain specific algorithm FLINT and tightly competes with MCSMUS. We recommend ReMUS for a use in an arbitrary constraint domain.

Part II

MUS and MSS/MCS Enumeration in the Boolean CNF Domain

5
Boolean CNF MUS Enumeration

In this chapter, we focus on the instance of MSMPs where the input set C is a set of Boolean clauses (i.e., a Boolean formula in CNF) and the monotone predicate $\mathbf{P}$ is the standard Boolean unsatisfiability. Therefore, the minimal subsets of C of interest are called *minimal unsatisfiable subsets (MUSes)*. In particular, this chapter describes our MUS enumeration algorithm, called UNIMUS [Bendík and Černá, 2020c], that is tailored for this constraint domain and that extensively exploits specific properties of Boolean clauses.

Minimal unsatisfiable subsets (over Boolean clauses) find applications in various areas of computer science such as Boolean function bi-decomposition [Chen and Marques-Silva, 2011], formal equivalence checking [Cohen et al., 2010], circuit error diagnosis [Han and Lee, 1999], type debugging in Haskell [Stuckey et al., 2003], counterexample guided abstraction refinement [Andraus et al., 2007], and many others [Mu, 2019, Arif et al., 2016, Jannach and Schmitz, 2016, Ivrii et al., 2016, Hunter and Konieczny, 2008].

The reason for identifying MUSes of the input set C of clauses is usually to somehow analyze the unsatisfiability of C. In some applications, the goal is to identify just a single MUS. In other applications, e.g. [Hunter and Konieczny, 2008, Mu, 2019, Andraus et al., 2007], it is desirable to enumerate several or even all MUSes of C. The more MUSes are identified, the better insight into the unsatisfiability is provided. However, recall that there can be in general up to exponentially many MUSes w.r.t. $|C|$ which often makes the complete MUS enumeration practically intractable. Consequently, contemporary MUS enumeration algorithms, e.g. [Liffiton et al., 2016, Bacchus and Katsirelos, 2015, 2016, Narodytska et al., 2018], enumerate MUSes gradually, i.e. one by one, and they attempt to identify as many MUSes as possible within a given time limit.

Naturally, the problem of MUS identification subsumes checking subsets of C for satisfiability. Since these checks are quite expensive (NP-complete), the efficiency of an MUS enumeration algorithm is often tightly connected to the number of satisfiability checks performed to identify each single MUS. Our MUS enumeration algorithm that we present in this section, i.e., UNIMUS, is much more frugal in the number of performed satisfiability checks than other contemporary MUS enumeration algorithms, which allows it to enumerate MUSes much faster.

The rest of this chapter is organized as follows. Section 5.1 introduces the notation that is specific for this chapter. Subsequently, Section 5.2 gives a detailed description of our algorithm. Other contemporary MUS enumeration solutions are described in Section 5.3. Finally, Section 5.4 provides experimental evaluation with other contemporary Boolean CNF MUS enumeration tools.

5.1 Notation

Throughout this chapter, the input set C is a set of Boolean clauses, and the monotone predicate $\mathbf{P}$ is the standard Boolean unsatisfiability. We simply say that a subset N of C is either *satisfiable* or *unsatisfiable*. Hence, the $\mathbf{P}_1$-minimal subsets are *minimal unsatisfiable subsets (MUSes)* and the $\mathbf{P}_0$-maximal subsets are *maximal satisfiable subsets (MSSes)*. Critical/conflicting elements for a subset of C are called *critical/conflicting clauses*, i.e., a clause $c \in N$ is critical for an unsatisfiable set N iff $N \setminus \{c\}$ is satisfiable, and a clause $d \notin M$ is conflicting for a satisfiable set M iff $M \cup \{d\}$ is unsatisfiable.

To check a subset of C for satisfiability, we employ a SAT solver. Moreover, given a subset N of C, we assume that the SAT solver is able to return either an *unsat core* of N when N is unsatisfiable, or a model and the corresponding *model extension* of N when N is satisfiable[1]

[1] Recall that an unsat core of N is an unsatisfiable subset of N. Similarly, given a satisfiable subset N of C and a model π of N (provided by a SAT solver), the model extension of N w.r.t. π is the satisfiable superset E of N defined as $E = \{c \in C \mid \pi \models c\}$ (Definition 2.4). Note that based on our empirical experience, the unsat core (model extension) is often very close to an MUS (MSS) of C.

We use the terminology of *unexplored subsets* of C when talking about the subsets whose satisfiability has not yet been determined by our algorithm, and we write `Unexplored` to denote the set of all unexplored subsets. Satisfiable unexplored subsets are called *s-seeds*, and unsatisfiable unexplored subsets are called *u-seeds*. The set `Unexplored` is maintained via the symbolic representation based on the formula $map^{+} \wedge map^{-}$ (described in Section 2.3.2).

5.2 Algorithm

UNIMUS is based on the seed-shrink scheme (as described in Section 4.2), i.e., to find each single MUS, UNIMUS first identifies a u-seed and then shrinks the u-seed into an MUS. It employs a novel shrinking procedure. Moreover, it employs two novel approaches for finding u-seeds. One of the approaches is based on the same idea as contemporary seed-shrink algorithms: to find a u-seed, UNIMUS is repeatedly picking and checking for satisfiability (via a SAT solver) an unexplored subset until it identifies a u-seed. The novelty is in the choice of unexplored subsets to be checked. Briefly, UNIMUS maintains a *base* B and a *search-space* $\texttt{Unex}_B = \{X \mid X \in \texttt{Unexplored} \wedge X \subseteq B\}$ that is induced by the base. UNIMUS searches for u-seeds only within $\texttt{Unex}_B$. The base B (and thus $\texttt{Unex}_B$) is maintained in a way that allows identifying u-seeds with performing only a few satisfi-

```
input : an unsatifiable set $C$ of Boolean clauses
output: all MUSes of $C$
1 Unexplored ← $\mathcal{P}(C)$                              // a global variable
2 $B \leftarrow \emptyset$
3 while Unexplored $\neq \emptyset$ do
4   |  $B \leftarrow$ refine($B$)                               // Algorithm 5.3
5   |  UNIMUSCore($B$)                                          // Algorithm 5.2
```

Algorithm 5.1: The Boolean CNF MUS enumeration algorithm UNIMUS.

ability checks. Moreover, the u-seeds are very close to MUSes and thus relatively easy to shrink.

Our other approach for finding u-seeds is based on a fundamentally different principle. Instead of checking unexplored subsets for satisfiability via a SAT solver, UNIMUS *deduces* that some unexplored subsets are unsatisfiable. The deduction is based on already identified MUSes and it is very cheap (polynomial).

5.2.1 *Main Procedure*

UNIMUS (Algorithm 5.1) first initializes Unexplored to $\mathcal{P}(C)$ and the base B to $\emptyset$. Then, it in a while-loop repeats two procedures: refine that updates the base B, and UNIMUSCore that identifies MUSes in the search-space $\texttt{Unex}_B = \{X \mid X \in \texttt{Unexplored} \land X \subseteq B\}$. The algorithm terminates once Unexplored $= \emptyset$.

UNIMUSCore (Algorithm 5.2) works iteratively. Each iteration starts by picking a *maximal element* N of $\texttt{Unex}_B$, i.e., $N \in \texttt{Unex}_B$ such that $N \cup \{c\} \notin \texttt{Unex}_B$ for every $c \in B \backslash N$. Subsequently, a procedure isSAT is used to determine, via a SAT solver, the satisfiability of N. Moreover, in dependence of N's satisfiability, isSAT returns either an unsat core K or a model extension E of N. If N is unsatisfiable, the algorithm shrinks the core K into an MUS K_{mus} and removes from Unexplored all subsets and all supersets of K_{mus}. Subsequently, a procedure replicate is invoked which attempts to identify additional MUSes in $\texttt{Unex}_B$.

In the other case, when N is satisfiable, we remove all subsets of E from Unexplored[2]. Then, N is used to guide the algorithm into a search-space with more minable critical clauses. In particular, since N was a maximal unexplored subset of B, then for every $c \in B \backslash N$ the set $N \cup \{c\}$ is explored and thus unsatisfiable (Observation 2.15). Consequently, every clause $c \in B \backslash N$ is critical for $N \cup \{c\}$ and especially for every u-seed contained in $N \cup \{c\}$. Moreover, all these critical clauses are minable critical as all subsets of N were removed from Unexplored. If $N \cup \{c\} = B$ then c is minable critical for every u-seed in the current search-space. Otherwise, if $|B \backslash N| > 1$, we recursively call UNIMUSCore with a base $B' = N \cup \{c\}$ for every $c \in B \backslash N$.

[2] Note that E might outreach the current search-space $\texttt{Unex}_B$ and thus prune search-spaces that will be processed by future calls of UNIMUSCore.

The procedures refine, replicate, and shrink are described in Sections 5.2.2, 5.2.3, and 5.2.4, respectively. All procedures of UNIMUS follow several rules about Unexplored. First, all operations over

```
1  while $\{X \mid X \in \texttt{Unexplored} \wedge X \subseteq B\} \neq \emptyset$ do
2      $N \leftarrow$ a maximal element of $\{X \mid X \in \texttt{Unexplored} \wedge X \subseteq B\}$          // $\texttt{Unex}_B$
3      $(sat?, E, K) \leftarrow \textsf{isSAT}(N)$        // $K$ is an unsat core or $E$ is a model extension of $N$
4      if not $sat?$ then
5          $K_{mus} \leftarrow \textsf{shrink}(K)$                          // see Section 5.2.4
6          output $K_{mus}$
7          $\texttt{Unexplored} \leftarrow \texttt{Unexplored} \setminus \{X \mid X \subseteq K_{mus} \vee X \supseteq K_{mus}\}$
8          $\textsf{replicate}(K_{mus}, N)$                               // Algorithm 5.4
9      else
10         $\texttt{Unexplored} \leftarrow \texttt{Unexplored} \setminus \{X \mid X \subseteq E\}$
11         if $|B \setminus N| > 1$ then
12             for $c \in B \setminus N$ do  $\textsf{UNIMUSCore}(N \cup \{c\})$
```

Algorithm 5.2: UNIMUSCore(B)

Unexplored comply with the four rules R1-R4 we defined in Section 2.3.1. Second, Unexplored is *global*, i.e., shared by the procedures. Third, we remove an unsatisfiable set from Unexplored only if the set is a superset of an explicitly identified MUS. Consequently, no MUS can be removed from Unexplored without being explicitly identified. Thus, when Algorithm 5.1 terminates (i.e., Unexplored = $\emptyset$), it is guaranteed that all MUSes were explicitly identified.

Heuristics According to the above description, UNIMUSCore terminates once all subsets, and especially all MUSes, of B become explored. However, based on our empirical experience, UNIMUSCore can get into a situation such that there is a lot of s-seeds in $\texttt{Unex}_B$ but only a few or even no u-seed. Consequently, the MUS enumeration can get stuck for a while. To prevent such a situation, we track the number of subsequent iterations of UNIMUSCore in which the set N was satisfiable. If there are 5 such subsequent iterations, we terminate the current recursive call of UNIMUSCore, i.e., we either backtrack to a parent call of UNIMUSCore or return to the main procedure (Algorithm 5.1).

5.2.2 *The Base and the Search-Space*

The base B is modified in two situations. The first situation is the case of recursive calls of UNIMUSCore which was described in the previous section. Here, we describe the second situation which is an execution of the procedure refine before each top-level call of UNIMUSCore. The goal is to identify a base B such that u-seeds in $\texttt{Unex}_B$ can be easily found and are relatively easy to shrink.

Our approach exploits the union $\texttt{UMUS}_C$ of all MUSes of C. Assume that we set B to $\texttt{UMUS}_C$. Since every MUS of C is contained in $\texttt{UMUS}_C$, the induced search-space $\texttt{Unex}_B$ would contain all MUSes. Moreover, compared to the whole C, the cardinality of $\texttt{UMUS}_C$ can be relatively small, and thus the u-seeds in $\texttt{Unex}_B$ would be easy to shrink. Unfortunately, a recent study [Mencía et al., 2019] shows that computing $\texttt{UMUS}_C$ is often practically intractable even for small in-

```
while Unexplored $\neq \emptyset$ do
    $T \leftarrow$ a maximal element of Unexplored
    $(sat?, E, K) \leftarrow \mathsf{isSAT}(T)$  // we use only the unsat core $K$ here ($T$ is unsat.  or an MSS)
    if not $sat?$ then
        $K_{mus} \leftarrow \mathsf{shrink}(K)$                    // see Section 5.2.4
        output $K_{mus}$
        Unexplored $\leftarrow$ Unexplored$\setminus\{X \mid X \subseteq K_{mus} \vee X \supseteq K_{mus}\}$
        return $B \cup K_{mus}$
    else Unexplored $\leftarrow$ Unexplored$\setminus\{X \mid X \subseteq T\}$
return $B$
```

Algorithm 5.3: refine(B)

put formulas. Thus, instead of initially computing $\mathtt{UMUS}_C$, we use as the base B just an under-approximation of $\mathtt{UMUS}_C$. Initially, we set B to $\emptyset$ (Algorithm 5.1, line 2), and in each call of refine we attempt to refine (enlarge) the under-approximation. Eventually, B becomes $\mathtt{UMUS}_C$ and, thus, eventually, the search-space will contain all so far unexplored MUSes of C.

The procedure refine (Algorithm 5.3) attempts to enlarge B with an unexplored MUS. In each iteration, it picks a maximal unexplored subset T, i.e., $T \in$ Unexplored such that $T \cup \{c\} \notin$ Unexplored for every $c \in C\setminus T$. Then, T is checked for satisfiability via the procedure isSAT. If T is unsatisfiable, then isSAT also returns an unsat core K of T, refine shrinks the core K into an MUS K_{mus} and based on K_{mus} updates the set Unexplored. Subsequently, refine terminates and returns an updated base $B' = B \cup K_{mus}$. Otherwise, if T is satisfiable, refine removes all subsets of T from Unexplored and continues with a next iteration.

There is no guarantee that each call of refine indeed enlarges B. One possibility is the corner case when all MUSes are already explored, but there are some s-seeds left. Another possibility is that the search-space $\mathtt{Unex}_B$ was not completely explored in the last call of UNIMUSCore due to the preemptive termination heuristic. Thus, refine might identify an MUS that is a subset of B. Yet, we have the guarantee that UNIMUS eventually finds all MUSes since it cannot terminate before all subsets of C become explored. Also, note that the procedure refine is very similar to an MUS enumeration algorithm MARCO [Liffiton et al., 2016]. The difference is that refine finds only a single unexplored MUS whereas MARCO finds them all (see Section 4.5 for details on MARCO).

5.2.3 MUS Replication

We now describe the procedure replicate(K_{mus}, N) that based on an identified MUS K_{mus} of N attempts to identify additional unexplored MUSes. The procedure follows the seed-shrink scheme, i.e., to find an MUS, it first identifies a u-seed and then shrinks it to an MUS. However, contrary to existing seed-shrink algorithms which identify

```
$\mathcal{M} \leftarrow \{K_{mus}\}$; $rStack \leftarrow \langle K_{mus} \rangle$
while $rStack$ is not empty do
    $M \leftarrow rStack.pop()$
    for $c \in M$ do
        if $c$ is minable critical for $N$ then continue
        $S \leftarrow$ propagate$(M, c, N, \mathcal{M})$                    // Algorithm 5.5
        if $S$ is null then continue
        $S_{mus} \leftarrow$ shrink$(S)$                                    // see Section 5.2.4
        output $S_{mus}$
        Unexplored $\leftarrow$ Unexplored$\setminus\{X \mid X \subseteq S_{mus} \vee X \supseteq S_{mus}\}$
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{S_{mus}\}$
        $rStack.push(S_{mus})$
```

Algorithm 5.4: replicate(K_{mus}, N)

u-seeds via a SAT solver, replicate identifies u-seeds with a cheap (polynomial) deduction technique; we call the technique *MUS replication*.

Each call of replicate possibly identifies several unexplored MUSes and all these MUSes are subsets of N. Note that since N is a subset of the base B in Algorithm 5.2, all MUSes identified by replicate are contained in the search-space $\texttt{Unex}_B$. Also, note that when replicate is called, K_{mus} is the only explored MUS of N (since N was a u-seed that we shrunk to K_{mus}). In the following, we will use $\mathcal{M}$ to denote the set of all explored MUSes of N, i.e., initially, $\mathcal{M} = \{K_{mus}\}$.

Main Procedure The main procedure of replicate, shown in Algorithm 5.4, maintains two data-structures: the set $\mathcal{M}$ and a stack *rStack*. The computation starts by initializing both $\mathcal{M}$ and *rStack* to contain the MUS K_{mus}. The rest of replicate is formed by two nested loops. In each iteration of the outer loop, replicate pops an MUS M from the stack. In the nested loop, M is used to identify possibly several unexplored MUSes. In particular, for each clause $c \in M$ the algorithm attempts to identify a u-seed S such that $M\setminus\{c\} \subsetneq S \subseteq N\setminus\{c\}$. Observe that if c is minable critical for N then such a u-seed cannot exist; thus, we skip such clauses. The attempt to find such S is carried out by a procedure propagate. If propagate fails to find the u-seed, the inner loop proceeds with a next iteration. Otherwise, the u-seed S is shrunk into an MUS S_{mus} and the set `Unexplored` is appropriately updated. The iteration is concluded by adding S_{mus} to $\mathcal{M}$ and also pushing S_{mus} to *rStack*, i.e., each identified MUS is used to possibly identify additional MUSes. The computation terminates once *rStack* becomes empty.

Propagate The procedure propagate is based on the well-known concepts of *backbone literals* and *unit propagation* [Bollobás et al., 2001, Kilby et al., 2005]. Given a formula P, a literal l is a *backbone literal* of P iff every model of P satisfies $\{l\}$. If P is unsatisfiable then every literal is a backbone literal of P. A *backbone* of P is a set of backbone literals of P. If A is a backbone of P then A is also a backbone of every superset of P. A clause d is a *unit clause* iff $|d| = 1$. Note that if d is a unit clause of P then the literal $l \in d$ is a backbone literal of P.

Given a backbone A of P, the *backbone propagation* can simplify P and possibly show that P is unsatisfiable. In particular, for every $l \in A$ and every clause $d \in P$ such that $\neg l \in d$, we remove the literal $\neg l$ from d (since no model of P can satisfy $\neg l$). If a new unit clause emerges during the propagation, the backbone literal that forms the unit clause will be also propagated. If the propagation reduces a clause to an empty clause, then the original P is unsatisfiable.

The procedure `propagate` employs backbone propagation to identify a u-seed S such that $M \backslash \{c\} \subsetneq S \subseteq N \backslash \{c\}$. Observe that since M is unsatisfiable and $M \backslash \{c\}$ is satisfiable, then the set $A = \{\neg l \,|\, l \in c\}$ is a backbone of every such S. Thus, one can pick such S and attempt to show, via propagating A, that S is unsatisfiable. However, there are too many such S to choose from. Moreover, we need to guarantee that we find S that is both unsatisfiable and unexplored. Thus, instead of fixing a particular S and then trying to show its unsatisfiability, we attempt to gradually build such S. Initially, we set S to $M \backslash \{c\}$ and we step-by-step add clauses from $N \backslash \{c\}$ to S. The addition of the clauses is driven by a currently known backbone A of S and also by the set $\mathcal{M}$ of explored MUSes to ensure that the resulting S is a u-seed.

Proposition 5.1. *For every* ***unsatisfiable*** *S, $S \subseteq N \backslash \{c\}$, it holds that $S \in$ `Unexplored` (i.e., S is a u-seed) if and only if $\forall_{X \in \mathcal{M}} S \not\supseteq X$.*

Proof. In UNIMUS, we remove from `Unexplored` only unsatisfiable sets that are supersets of explored MUSes and all explored MUSes of N are stored in $\mathcal{M}$. □

Proposition 5.1 shows which clauses *can be added* to the initial S while ensuring that if we finally obtain an unsatisfiable S, then the final S will be unexplored. Note that the initial $S = M \backslash \{c\}$ trivially satisfies $\forall_{X \in \mathcal{M}} M \backslash \{c\} \not\supseteq X$ since it is satisfiable (M is an MUS). In the following, we show which clauses *should be added* to S to eventually make it unsatisfiable.

Definition 5.1 (operation $\backslash\backslash$). *Let d be a clause and A a set of literals. The the binary operation $d \backslash\backslash A$ produces the clause $d \backslash\backslash A = \{l \mid l \in d$ and $\neg l \notin A\}$.*

Definition 5.2 (units, violated). *Let S be a set such that $M \backslash \{c\} \subsetneq S \subseteq N \backslash \{c\}$, and let A be a backbone of S. We define the following sets:*

$$units(S, A) = \{d \in N \backslash \{c\} \mid \forall_{X \in \mathcal{M}} S \cup \{d\} \not\supseteq X \wedge |d \backslash\backslash A| = 1\}$$
$$violated(S, A) = \{d \in N \backslash \{c\} \mid \forall_{X \in \mathcal{M}} S \cup \{d\} \not\supseteq X \wedge |d \backslash\backslash A| = 0\}$$

Informally, a clause $d \in N \backslash \{c\}$ belongs to *units*(S, A) (*violated*(S, A)) if the propagation of A would simplify d to a unit clause (empty clause) and, simultaneously, $d \in S$ or d can be added to S in a harmony with Proposition 5.1.

```
$S \leftarrow M \backslash \{c\}$; $A \leftarrow \{\neg l \mid l \in c\}$; $H \leftarrow \{c\}$
while $units(S,A) \backslash H \neq \emptyset$ and $violated(S,A) = \emptyset$ do
    $d \leftarrow$ choose $d \in units(S,A) \backslash H$
    $S \leftarrow S \cup \{d\}$; $H \leftarrow H \cup \{d\}$; $A \leftarrow A \cup d \setminus\!\setminus A$
if $violated(S,A) = \emptyset$ then return null
else
    $d \leftarrow$ choose $d \in violated(S,A)$
    return $S \cup \{d\}$
```

Algorithm 5.5: propagate$(M, c, N, \mathcal{M})$

Proposition 5.2. *For every S such that $M \backslash \{c\} \subsetneq S \subseteq N \backslash \{c\}$, a backbone A of S, and a clause $d \in units(S,A)$, it holds that $A \cup d \setminus\!\setminus A$ is a backbone of $S \cup \{d\}$.*

Proof. Assume that $d = \{l, l_0, \dots, l_k\}$ where $\{\neg l_0, \dots, \neg l_k\} \subseteq A$ and $l = d \setminus\!\setminus A$. Since A is a backbone of S then every model of S satisfies $\{\{\neg l_0\}, \dots, \{\neg l_k\}\}$. Consequently, every model of $S \cup \{d\}$ satisfies $\{l\}$. □

Proposition 5.3. *For every S such that $M \backslash \{c\} \subsetneq S \subseteq N \backslash \{c\}$, a backbone A of S, and a clause $d \in violated(S,A)$, it holds that $S \cup \{d\}$ is unsatisfiable.*

Proof. Assume that $d = \{l_0, \dots, l_q\}$. Since $d \in violated(S,A)$, then $\{\neg l_0, \dots, \neg l_q\} \subseteq A$. Furthermore, A is a backbone of S, and thus every model of S satisfies $\{\{\neg l_0\}, \dots, \{\neg l_q\}\}$, i.e., no model of S satisfies d. □

The procedure propagate, shown in Algorithm 5.5, maintains three data structures: the sets S and A, and an auxiliary set H for storing clauses that were used to enlarge A. Initially, $S = M \backslash \{c\}$, $A = \{\neg l \mid l \in c\}$ and $H = \{c\}$. In each iteration, propagate picks a clause $d \in units(S,A) \backslash H$ and, based on Proposition 5.2, adds d to S and to H, and the literal of $d \setminus\!\setminus A$ to A. The loop terminates once there is no more backbone literal to propagate ($units(S,A) \backslash H = \emptyset$), or once $violated(S,A) \neq \emptyset$. If $violated(S,A) = \emptyset$, propagate failed to find a u-seed. Otherwise, propagate picks a clause $d \in violated(S,A)$ and returns the u-seed $S \cup \{d\}$.

Finally, note that backbone propagation is cheap (polynomial) but it is not a *complete* technique for deciding satisfiability. Consequently, it can happen that there is a u-seed S, $M \backslash \{c\} \subsetneq S \subseteq N \backslash \{c\}$, but MUS replication fails to find it.

5.2.4 Shrink

UNIMUS implements the seed-shrink scheme (Section 4.2), and thus, the shrinking of a u-seed N is carried out via a black-box single MUS extraction subroutine. This means that the only output of the shrinking is an MUS of N; there is no other side effect on the overall MUS enumeration. The advantage of the black-box approach is

```
   input : a u-seed $N$
   output: a set $crits$ of clauses that are critical for $N$
 1 $crits \leftarrow$ collect all minable critical clauses for $N$
 2 $Q \leftarrow crits$
 3 while $Q \neq \emptyset$ do
 4     $c \leftarrow$ pick $c \in Q$
 5     $Q \leftarrow Q \setminus \{c\}$
 6     for $l \in c$ do
 7         $M \leftarrow \{d \in N \mid \neg l \in d\}$
 8         if $|M| = 1$ and $M \cap crits = \emptyset$ then
 9             $crits \leftarrow crits \cup M$
10             $Q \leftarrow Q \cup M$
11 return $crits$
```

Algorithm 5.6: The critical extension technique.

that we can use any available single MUS extraction tool to implement the shrinking, and especially, any future advance in a single MUS extraction can be immediately reflected in the performance of UNIMUS.

There exist several publicly available single MUS extraction tools (e.g., [Belov and Marques-Silva, 2012, Belov et al., 2014, Nadel et al., 2014, Bacchus and Katsirelos, 2015]). Given an unsatisfiable set N, contemporary extractors also allow the user to provide a set of critical clauses for N, since prior knowledge of the critical clauses can significantly speed up the extraction. Thus, before shrinking N, we attempt to identify a set *crits* of clauses that are critical for N and pass them to the single MUS extractor together with N.

As already discussed in the context of domain agnostic MUS enumeration algorithms (Chapter 4), we can use the set `Unexplored` to collect minable critical clauses for N. In UNIMUS, we indeed collect all the minable critical clauses. However, we identify more than just *minable* critical clauses for N. We introduce a technique that, based on the minable critical clauses for N, can cheaply *deduce* that some other clauses are critical for N. We call the deduction technique *critical extension* and it is based on the following observation.

Proposition 5.4. *Let N be a u-seed, $c \in N$ a critical clause for N, and $l \in c$ a literal of c. Moreover, let $M \subseteq N$ be the set of all clauses of N that contain the literal $\neg l$. If $|M| = 1$ then the clause $d \in M$ is critical for N, i.e. $N \setminus \{d\}$ is satisfiable.*

Proof. Assume that $N \setminus \{d\}$ is unsatisfiable. Since c is critical for N, then c is critical also for $N \setminus \{d\}$. Thus, $N \setminus \{c, d\}$ is satisfiable and every its model satisfies $\{\neg l\}$ (as $l \in c$) which contradicts that $\neg l$ is contained only in d. □

The critical extension technique (Algorithm 5.6) takes as an input a u-seed N and outputs a set *crits* of clauses that are critical for N. The algorithm starts by collecting (see Section 2.3.2) all minable critical clauses for N and stores them to *crits* and also to an auxiliary set Q. The rest of the computation works iteratively. In each iteration, the algorithm picks and removes a clause c from Q and employs Proposition 5.4 on c. In particular, for each literal $l \in c$, the algorithm builds the set $M = \{d \in N \mid \neg l \in d\}$. If M contains only a single clause, say d, and $d \notin crits$, then d is a new critical clause for N and thus it is added to *crits* and to Q. The computation terminates once Q becomes empty.

Our technique is similar to *model rotation* [Belov and Marques-Silva, 2011, Bacchus and Katsirelos, 2015] which identifies additional critical clauses based on a critical clause c of N and a model of $N \backslash \{c\}$. The difference is that we do not need the model. Another approach [Wieringa, 2012] that also does not need the model is based on *rotation edges* in a *flip graph* of C.

5.3 Related Work

MUS enumeration was extensively studied in the past decades and many various algorithms were proposed, e.g., [Hou, 1994, Han and Lee, 1999, de la Banda et al., 2003, Bailey and Stuckey, 2005, Stern et al., 2012, Liffiton and Sakallah, 2008, Previti and Marques-Silva, 2013, Liffiton and Malik, 2013, Bendík et al., 2016b, 2018b, Narodytska et al., 2018, Bacchus and Katsirelos, 2015, 2016]. In the following, we briefly describe contemporary MUS enumeration approaches.

FLINT [Narodytska et al., 2018] computes MUSes in *rounds* and each round consists of two phases: *relaxing* and *strengthening*. In the relaxing phase, the algorithm starts with an unsatisfiable formula U and weaknesses it by iteratively relaxing its unsat core until it gets a satisfiable formula S. The intermediate unsat cores are shrunk to MUSes (via an external single MUS extractor). The resulting satisfiable formula S is passed to the second phase, where the formula is again strengthened to an unsatisfiable formula that is used in the next round as an input for the relaxing phase.

Another state-of-the-art MUS enumerators are MARCO [Liffiton et al., 2016] and ReMUS [Bendík et al., 2018b] (as described in Chapter 4). Although these two algorithms are domain agnostic, i.e., not tailor to the Boolean CNF domain, they are still quite efficient in this domain. Same as UNIMUS, MARCO and ReMUS are based on the seed-shrink scheme and thus the major difference between the algorithms is in the way they identify u-seeds for shrinking. Recall that MARCO is iteratively picking and checking for satisfiability a maximal unexplored subset of C, until it finds a u-seed. Since unsatisfiable subsets of C are naturally more concentrated among the larger subsets, MARCO usually performs only a few checks to find the u-seed. However, large u-seeds are generally hard to shrink. Thus, the efficiency of MARCO crucially depends on the capability of the SAT solver to provide a reasonably small unsat core of the u-seed. ReMUS, on contrary to MARCO, tends to identify u-seeds that are relatively small and thus easy to shrink, by recursively reducing the search-space where it looks for u-seeds. In some sense, the use of a recursive search-space in ReMUS is similar to the use of the search-space $\mathtt{Unex}_B$ in UNIMUS; however, whereas ReMUS gradually reduces the search-space, UNIMUS on the other hand gradually enlarges the search-space. The biggest difference between ReMUS, MARCO, and UNIMUS is that UNIMUS is not a domain agnostic algorithm and hence it can directly exploit specific properties of Boolean clauses. Especially, note that UNIMUS is the first seed-shrink algorithm that is able to identify u-seeds in a polynomial time (via the MUS replication), i.e., without a SAT solver. MARCO and ReMUS, on the other hand, are domain agnostic and hence cannot rely on any domain specific properties.

MCSMUS [Bacchus and Katsirelos, 2016] is an MUS enumeration algorithm that is tailored for the Boolean CNF domain. Same as MARCO, MCSMUS searches for u-seeds among maximal unexplored subsets and shrinks them to MUSes. However, contrary to MARCO, MCSMUS implements the shrinking via a custom procedure. The shrinking procedure works in a recursive manner and can possibly identify multiple MUSes, i.e., multiple MUSes originate from a single u-seed. Moreover, the shrinking procedure of MCSMUS fully shares information and works in synergy with the overall MUS enumeration algorithm. For example, all satisfiable subsets of C that are identified during the shrinking are remembered by the algorithm and then used to speed up the subsequent shrinks.

5.4 *Experimental Evaluation*

We implemented UNIMUS using miniSAT [Eén and Sörensson, 2003] to maintain the set `Unexplored`, CaDiCaL [Biere, 2018] as the SAT solver for checking subsets of C for satisfiability, and a single MUS extractor subroutine from the MUS enumeration tool MCSMUS [Bacchus and Katsirelos, 2016] to implement the shrinking. Our tool is publicly available at:

https://github.com/jar-ben/unimus

Here, we provide results of an experimental comparison with with four contemporary MUS enumeration solutions: MARCO [Liffiton et al., 2016][3], MCSMUS [Bacchus and Katsirelos, 2016][4], FLINT [Narodytska et al., 2018][5], and ReMUS [Bendík et al., 2018b][6]. As benchmarks, we used the collection of 291 CNF formulas from the MUS track of the SAT 2011 Competition[7]. This collection is standardly used in MUS related papers, including the papers that presented our four competitors. The formulas range in their size from 70 to 16 million constraints and use from 26 to 4.4 million variables.

All experiments were run using a time limit of 3600 seconds and computed on an AMD 16-Core Processor and 1 TB memory machine running Debian Linux. Complete results are available in the online appendix[8]. The comparison criteria are the number of identified MUSes within the time limit and the performance stability of

[3] MARCO is available at https://sun.iwu.edu/~mliffito/marco/

[4] MCSMUS is available at https://bitbucket.org/gkatsi/mcsmus/src

[5] FLINT was kindly provided to us by its author, Nina Narodytska.

[6] ReMUS is implemented in the tool MUST available at https://github.com/jar-ben/mustool

[7] http://www.cril.univ-artois.fr/SAT11/

[8] https://www.fi.muni.cz/~xbendik/phdThesis/

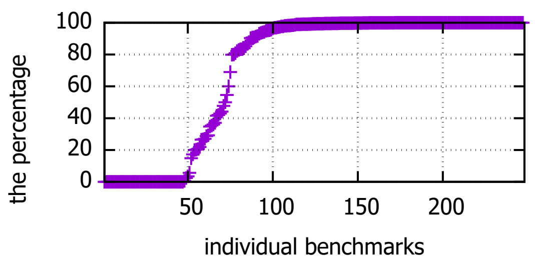


Figure 5.1: Percentage of MUSes found by MUS replication.

the algorithms in time. Moreover, we examine in more detail the manifestation of MUS replication in UNIMUS.

5.4.1 *Manifestation of MUS Replication*

MUS replication is a crucial part of UNIMUS as, to the best of our knowledge, it is the first existing technique that identifies u-seeds in polynomial time. Therefore, we are interested in what is the percentage of u-seeds, and thus MUSes, that UNIMUS identifies via the MUS replication. Figure 5.1 shows this percentage (y-axis) for individual benchmarks (x-axis); the benchmarks are sorted by the percentage. We computed the percentage only for the 248 benchmarks where UNIMUS found at least 5 MUSes. Remarkably, in the case of 161 benchmarks, the percentage is higher than 90 percent, and in the case of 130 benchmarks, it is higher than 99 percent. Unfortunately, there are 49 benchmarks where MUS replication found no u-seed at all. Let us note that 40 of the 49 benchmarks are from the *same family* of benchmarks, called "fdmus". The MUS benchmarks from the SAT competition consist of multiple families and benchmarks in a family often have a very similar structure. Most of the families contain only a few benchmarks, however, there are several larger families and the "fdmus" family is by far the largest one.

5.4.2 *Number of Identified MUSes*

We now examine the number of identified MUSes by the evaluated algorithms on individual benchmarks within the time limit of 3600 seconds. In the case of 28 benchmarks, all the algorithms completed the enumeration, and thus found the same number of MUSes. Therefore, we focus here only on the remaining 263 benchmarks.

Figure 5.2 provides scatter plots that pair-wise compare UNIMUS with its competitors. Each point in the plot shows a result from a single benchmark. The x-coordinate of a point is given by the algorithm that labels the x-axis and the y-coordinate by the algorithm that labels the y-axis. The plots are in a log-scale and hence cannot show points with a zero coordinate, i.e., benchmarks where at least one algorithm found no MUS. Therefore, we lifted the points with a zero coordinate to the first coordinate. Moreover, we provide three numbers right/above/in the right corner of the plot, that show the number of points below/above/on the diagonal. For example,

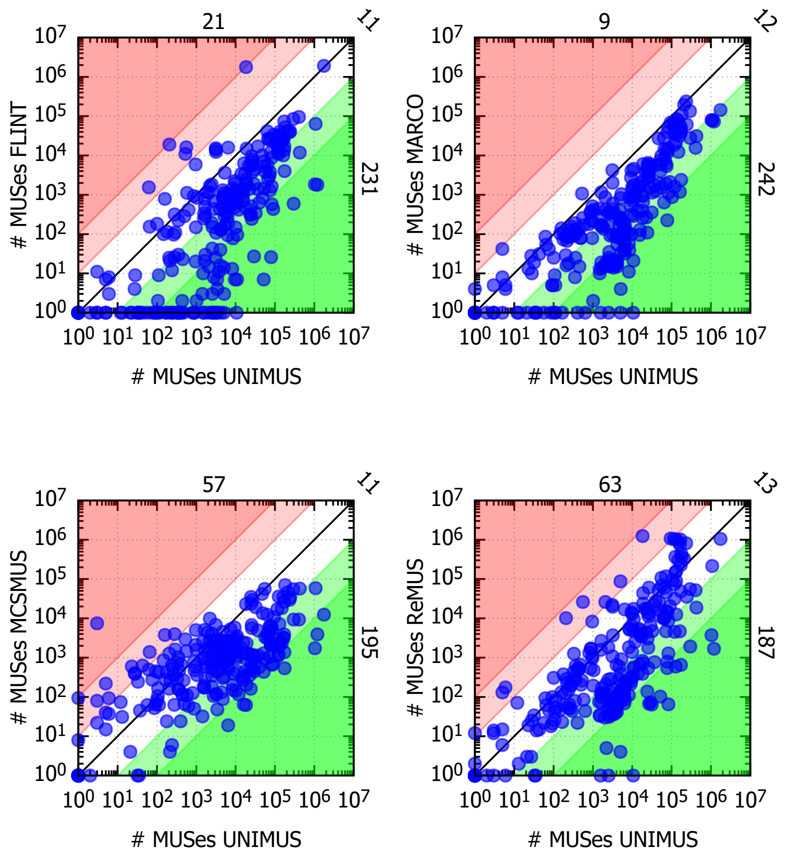


Figure 5.2: Scatter plots comparing the number of produced MUSes.

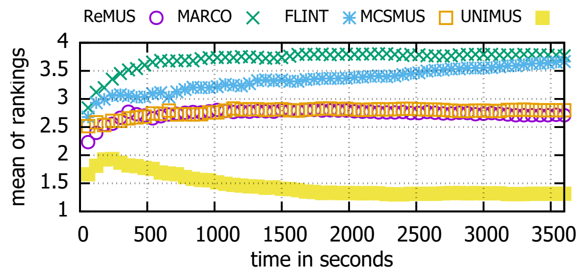


Figure 5.3: 5% truncated mean of rankings after each 60 seconds.

UNIMUS found more/less/equal number of MUSes than MARCO in case of 242/9/12 benchmarks. We also use green and red colors to highlight individual orders of magnitude (of 10).

Besides the pair-wise comparison, we examine also an overall ranking of the algorithms on individual benchmarks. In particular, assume that for a benchmark B both UNIMUS and ReMUS found 100 MUSes, MCSMUS found 80 MUSes, and MARCO and FLINT found 50 MUSes. In such a case, UNIMUS and ReMUS share the 1st (best) rank for B, MCSMUS is 3rd, and MARCO and FLINT share the 4th position. For each algorithm, we computed an arithmetic mean of the ranking on all benchmarks. To eliminate the effect of outliers (benchmarks with an extreme ranking), we computed the 5 percent truncated arithmetic mean, i.e., for each algorithm we discarded the 5 percent of benchmarks where the algorithm achieved the best and the worst ranking. Moreover, to capture the performance stability of the algorithms in time, we computed the mean for each subsequent 60 seconds of the computation. The results are in Figure 5.3.

We conclude that UNIMUS significantly dominates all its competitors. It maintained the best ranking during the whole time period. Moreover, its ranking was gradually improving towards the final value of 1.3. The closest, yet still very distant, competitors are ReMUS and MCSMUS who steadily maintained a ranking around 2.75. FLINT and MARCO achieved the final raking around 3.7. UNIMUS also dominated in the pair-wise comparison. It found more MUSes than all its competitors on an overwhelming majority of benchmarks and, remarkably, the difference was often several orders of magnitude. We believe that such a significant improvement is achieved both due to the MUS replication, which allows us to find u-seeds without a SAT solver, and the restriction to the local search-space $\texttt{Unex}_B$, which guarantees that the u-seeds are close to MUSes.

Finally, let us note that the performance of all the compared algorithms is also affected by the efficiency of their implementation, and mainly by the choice the underlying SAT solvers. As we have already said, UNIMUS uses CaDiCaL to perform the satisfiability checks. However, as witnessed in Figure 5.1, a UNIMUS found a vast majority of u-seeds via the MUS replication, i.e., without a SAT solver. Hence, UNIMUS performs satisfiability checks almost only during the shrinking, which is implemented by calling a single MUS extraction subroutine of MCSMUS, i.e., UNIMUS mostly uses the same SAT solver as MCSMUS. In particular, MCSMUS employs Glucose [Audemard and Simon, 2009] as the SAT solver. ReMUS and MARCO employ miniSAT. To perform the shrinking, ReMUS employs the single MUS extraction subroutine of MCSMUS, and MARCO and FLINT employ a single MUS extractor called muser2 [Belov and Marques-Silva, 2012]. Finally, we do not know which SAT solver is used by FLINT since we were provided only a pre-compiled binary of FLINT by its authors.

6 Boolean CNF MSS and MCS Enumeration

Same as in the previous chapter, here we focus on the instance of MSMPs where the input set C is a set of Boolean clauses (i.e., a Boolean formula in CNF) and the monotone predicate $\mathbf{P}$ is the standard Boolean unsatisfiability. The $\mathbf{P}_1$-minimal subsets of C are thus Minimal Unsatisfiable Subsets (MUSes), the $\mathbf{P}_0$-maximal subsets of C are Maximal Satisfiable Subsets (MSSes), and the complements of MSSes are Minimal Correction Subsets (MCSes). Whereas in the previous chapter we have focused on MUS enumeration, here our goal is to enumerate MSS/MCSes of the input formula.

Due to the massive improvements in SAT solving in the past two decades, Boolean formulas are nowadays widely used in almost all areas of computer science. Consequently, MSSes (or MCSes), as a natural concept for analyzing the unsatisfiability of a CNF formula, found many practical applications in recent years and new applications are still arising. For instance, MSSes (MCSes) can be used during the computation of minimal models of CNF formulas and model-based diagnosis [Ben-Eliyahu and Dechter, 1993], ontology debugging, and axiom pinpointing [Arif et al., 2015a]. Another application of MSSes emerges in the context of the maximum satisfiability (MaxSAT) problem. In particular, the MSSes with the maximum cardinality are the exact solutions of the MaxSAT problem, and the other MSSes can be at least used as good under-approximations of the exact MaxSAT solutions [Marques-Silva et al., 2013a]. Yet another application of MSSes (MCSes) emerge in the area of diagnosis; given an unsatisfiable formula C, the number of MSSes of C serves as a good diagnosis metric of C's *degree* of unsatisfiability [Thimm, 2018]. Finally, due to the minimal hitting set duality between MUSes and MCSes (Observation 2.6), an existence of an efficient MSS/MCS enumeration algorithm can pave the way for efficient complete MUS enumeration [Liffiton and Sakallah, 2008].

Several contemporary MSS enumeration algorithms are based on a *seed-grow* scheme, i.e., a dual scheme to the seed-shrink scheme for MUS enumeration (Section 4.2). That is, to find each single MSS, a *seed-grow* algorithm first identifies an s-seed, i.e., a satisfiable subset among the unexplored subsets. Subsequently, the s-seed is grown into an MSS. The exact ways of finding and growing the s-seeds differ for individual seed-grow algorithms. In this chapter, we present a novel MSS/MCS enumeration algorithm called RIME [Bendík and Černá, 2020b]. As opposed to existing seed-grow algorithms which find s-seeds by checking unexplored subsets for satisfiability via a SAT solver, RIME is often able to find s-seeds using cheap (polynomial) deduction technique. Moreover, RIME uses a novel growing procedure.

We experimentally compare RIME with three contemporary MSS (MCS) enumeration approaches on a standard collection of benchmarks. The results show that RIME significantly outperforms its competitors on a majority of the benchmarks in the terms of identified MSSes within a given time limit. Remarkably, RIME needs to perform much fewer calls to a SAT solver to identify individual MSSes than its competitors do. On average, RIME performs just 1.13 satisfiability checks per MSS, and in the case of many benchmarks, the average number of checks per MSS is even smaller than 1. The average number of performed satisfiability checks per MSS is even more impressive knowing that the problem of identifying a single MSS of a given formula C is in $FP^{NP}[log]$, i.e., a single MSS can be found using logarithmically many calls of a SAT solver w.r.t. $|C|$ [Janota and Marques-Silva, 2016].

This chapter is organized as follows. First, Section 6.1 defines the notation that is specific for this part of the thesis. Subsequently, Section 6.2 provides a detailed description of RIME. Section 6.3 discusses other existing MSS/MCS enumeration approaches, and finally Section 6.4 provides results of our experimental evaluation.

6.1 Notation

We use the same notation as in the previous chapter. In particular, throughout the whole chapter, C is used to denote the input set of Boolean clauses. We use the terminology of *unexplored subsets* and we write `Unexplored` to denote the set of all unexplored subsets of C (unexplored by our algorithm). Moreover, we obey the rules R1-R4 on manipulation with `Unexplored` as defined in Section 2.3.1. Unexplored unsatisfiable subsets are called u-seeds and unexplored satisfiable subsets are called s-seeds.

We assume that a SAT solver, given a subset N of C, is able to return either an unsat core $K \subseteq N$ of N, or a model π of N which can be used to build the corresponding model extension $E \supseteq N$ of N defined as $E = \{c \in C \,|\, \pi \models c\}$. Recall that the unsat core is unsatisfiable and the model extension is satisfiable.

Critical/conflicting elements (Definitions 2.8 and 2.9) for a subset of C are called critical/conflicting clauses. Furthermore, we work with *backbone literals* of a set $S \subseteq C$, i.e., literals that have to be satisfied by every model of S. Importantly, we exploit the following connection between conflicting clauses and backbone literals:

Proposition 6.1. *Let S be a subset of C and $c \in C \setminus S$ a conflicting clause for S, i.e., $S \cup \{c\}$ is unsatisfiable. Then the literals $\{\neg l \mid l \in c\}$ are backbone literals of every S', $S' \supseteq S$.*[1].

[1] Note that we exploited this observation also during the MUS replication in Section 5.2.3

Proof. By contradiction, assume that c is conflicting for S, however, a literal $\neg l$ such that $l \in c$ is not a backbone literal for S. Therefore, there is a model π of S that satisfies l, and hence $\pi \models S \cup \{l\}$ (which contradicts that c is conflicting for S). The fact that every backbone literal of S is also a backbone literal of every $S' \supseteq S$ follows from the fact that every model of such S' is also a model of S. □

6.2 Algorithm

Our MSS enumeration algorithm, called RIME, is based on a scheme that we call *seed-grow scheme*: to find each single MSS, we find an s-seed N and then we *grow* N into an MSS N_{mss} of C such that $N \subseteq N_{mss} \subseteq C$. The seed-grow scheme has been already used by several MSS enumeration algorithms, e.g., [Bailey and Stuckey, 2005, Stern et al., 2012, Liffiton et al., 2016, Bendík et al., 2016b]. In general, contemporary seed-grow algorithms identify an s-seed by iteratively picking and checking an unexplored subset for satisfiability, via a SAT solver, until they find an s-seed. Individual seed-grow algorithms vary in *which* and *how many* unexplored subsets are checked for satisfiability. Moreover, the algorithms vary in how exactly they grow the s-seeds. We provide more details on contemporary seed-grow algorithms and other MSS enumeration algorithms in Section 6.3. Naturally, satisfiable subsets of F are more concentrated among the smaller subsets, thus, it is easier (requiring fewer satisfiability checks) to find an s-seed among small unexplored subsets. On the other hand, the closer is an s-seed to an MSS, the easier is to grow the s-seed; thus it is worth to find a relatively large s-seed. Individual algorithms choose a different trade-off between the number of satisfiability checks performed to find the s-seed and the size of the s-seed.

RIME employs a novel growing procedure. Moreover, RIME alternates two novel techniques for finding s-seeds. One technique works on the same principle as the contemporary algorithms do: RIME repeatedly checks unexplored subsets for satisfiability until it finds an s-seed. The novelty is in the choice of unexplored subsets to be checked. Briefly, RIME maintains a *base* I_C, $I_C \subseteq C$, and a *search-space* $\mathcal{X}$ defined as $\mathcal{X} = \texttt{Unexplored} \cap \{N \mid I_C \subseteq N \subseteq C\}$, i.e. $\mathcal{X}$ consists of those unexplored subsets that contain the whole base. RIME searches for s-seeds in the search-space $\mathcal{X}$. The base is gradually updated in a way ensuring that s-seeds in the search-space can be easily found and grown, and that we eventually find all MSSes.

The other technique for finding s-seeds, called *MSS rotation*, is fundamentally different: we exploit the already explored MSSes to deduce that some unexplored subsets are satisfiable (i.e. s-seeds). The deduction is very cheap (polynomial) and does not involve the

```
input : an unsatisfiable set $C$ of Boolean clauses
output: all MSSes of $C$
1  Unexplored $\leftarrow \mathcal{P}(C) \setminus \{C\}$                                  // a global variable
2  $I_C \leftarrow C$
3  while Unexplored $\neq \emptyset$ do
4      $I_C \leftarrow$ refine($I_C$)                                                    // Algorithm 6.2
5      while Unexplored $\cap \{N \mid I_C \subseteq N \subseteq C\} \neq \emptyset$ do
6          $N \leftarrow$ a minimal element of Unexplored $\cap \{N \mid I_C \subseteq N \subseteq C\}$
7          $(sat?, E, K) \leftarrow$ isSAT($N$)         // $K$ is an unsat core or $E$ is a model extension of $N$
8          if $sat?$ then
9              $E_{mss} \leftarrow$ grow($E$)                                             // Algorithm 6.6
10             output $E_{mss}$
11             Unexplored $\leftarrow$ Unexplored$\setminus \{X \mid X \subseteq E_{mss} \vee X \supseteq E_{mss}\}$
12             $I_C \leftarrow$ rotate($E_{mss}, I_C$)                                     // Algorithm 6.3
13         else Unexplored $\leftarrow$ Unexplored$\setminus \{X \mid X \supseteq K\}$
```

Algorithm 6.1: The Boolean CNF MSS enumeration algorithm RIME.

usage of a SAT solver. In the following, we gradually provide a thorough description of all parts of RIME.

6.2.1 *Main Procedure*

The main procedure of our RIME is shown in Algorithm 6.1. Initially, the base I_C is set to C and Unexplored is set to $\mathcal{P}(C) \setminus \{C\}$ (we assume that C is unsatisfiable)[2]. The rest of the algorithm is formed by two nested while-loops. At the start of each iteration of the outer loop, the algorithm updates the base I_C via a procedure refine. Moreover, refine can possibly identify some MSSes. Subsequently, in the nested loop, the algorithm finds all unexplored MSSes in the search-space $\mathcal{X}$ = Unexplored $\cap \{N \mid I_C \subseteq N \subseteq C\}$. In particular, each iteration of the nested loop starts by picking a minimal element N of $\mathcal{X}$ where *minimal* means that $N \in \mathcal{X}$ and for all $f \in N$ it holds that $N \setminus \{f\} \notin \mathcal{X}$. Subsequently, N is processed by a procedure isSAT that determines the satisfiability of N (via a SAT solver) and moreover identifies either an unsat core K or a model extension E of N. If N is unsatisfiable, the algorithm removes all supersets of the unsat core K from Unexplored (since they are all unsatisfiable). In the other case, when N is satisfiable, the algorithm grows the model extension E into an MSS E_{mss} using a procedure grow. Moreover, the algorithm removes all subsets and all supersets of E_{mss} from Unexplored, since none of them can be another MSS. Then, RIME applies on E_{mss} the *MSS Rotation technique* (denoted by rotate) that tends to identify additional MSSes. The MSS rotation can also reduce the set Unexplored and update the base I_C. The inner loop terminates once $\mathcal{X} = \emptyset$. The outer loop terminates once Unexplored $= \emptyset$.

Detailed description of how the procedures refine, rotate, and grow work are provided in Sections 6.2.2, 6.2.3 and 6.2.4, respectively. For now, let us state some additional properties about the set Unexplored.

[2] Note that in the previous chapters, we did not require the input set C to be unsatisfiable. Here, we need it due to technical reasons. Also, note that based on the rules R1-R4 on manipulation with Unexplored (defined in Section 2.3.1), Unexplored should be initially set to the whole $\mathcal{P}(C)$. RIME follows the rules R1-R4. In particular, setting Unexplored to $\mathcal{P}(C) \setminus \{C\}$ can be performed in two steps: first, we set Unexplored to $\mathcal{P}(C)$ (in accordance to rule R1), and then we remove C from Unexplored (in accordance to the other rules).

First, `Unexplored` is a global variable, i.e., it is shared by all procedures of RIME. Second, in all the procedures we remove elements from `Unexplored` only in two situations. First, every time RIME identifies an MSS, it removes the MSS together with all subsets and all supersets of the MSS from `Unexplored`. Second, RIME can remove an unsatisfiable U, $U \subseteq C$, together with all supersets of U from `Unexplored`. Thus, no MSS can be removed from `Unexplored` without being explicitly identified. Furthermore, RIME follows the rules R1-R4 for manipulation with `Unexplored`, hence, by Observation 2.11, the following observation holds:

Observation 6.1. *The outer loop of Algorithm 6.1 terminates iff all MSSes and all MUSes have been explored (every $N \subseteq C$ is a subset or a superset of an MSS or an MUS of C, respectively).*

6.2.2 *The Base and the Search-Space*

Here we describe how we form and maintain the base I_C and thus the search-space $\mathcal{X} = \texttt{Unexplored} \cap \{N \mid I_C \subseteq N \subseteq C\}$. Naturally, it holds that the closer, in terms of cardinality, is an s-seed to an MSS, the easier it is to grow the s-seed to an MSS. Thus, we tend to keep in the search-space only s-seeds that are close to MSSes. We exploit the intersection $\texttt{IMSS}_C$ of all MSSes of C. Assume that we set I_C to $\texttt{IMSS}_C$; such a base has a nice property as stated in Proposition 6.2.

Proposition 6.2. *Let $I_C = \texttt{IMSS}_C$. Then for every MSS M of C such that $M \in \texttt{Unexplored}$ it holds that $M \in \mathcal{X} = \texttt{Unexplored} \cap \{N \mid I_C \subseteq N \subseteq C\}$.*

Proof. Since $\texttt{IMSS}_C$ is the intersection of all MSSes of C and $I_C = \texttt{IMSS}_C$, then $I_C \subseteq M$. □

It is not the case that such $\mathcal{X}$ consists only of MSSes; however, compared to the whole $\mathcal{P}(C)$, $\mathcal{X}$ is very dense in the terms of presented MSSes. Also, s-seeds in $\mathcal{X}$ are relatively close, in terms of cardinality, to MSSes and thus should be easy to grow to MSSes. Unfortunately, computing $\texttt{IMSS}_C$ is often practically intractable in a reasonable time (see the work [Mencía et al., 2019] on computing $\texttt{IMSS}_C$). Thus, RIME maintains an over-approximation of $\texttt{IMSS}_C$ as the base I_C, i.e. $\texttt{IMSS}_C \subseteq I_C \subseteq C$. Initially, we set I_C to C (Algorithm 6.1, line 2), and in each call of the procedure refine (Algorithm 6.1, line 4), we reduce I_C by removing at least a single clause of $I_C \setminus \texttt{IMSS}_C$ from I_C. Eventually, I_C becomes $\texttt{IMSS}_C$, and thus eventually the search-space $\mathcal{X}$ will contain all so far unexplored MSSes (Proposition 6.2).

We now describe the procedure refine. Let us by $\texttt{Unexplored}^i$ and I_C^i denote the values of `Unexplored` and I_C when Algorithm 6.1 calls refine. The procedure is based on the following three observations.

Proposition 6.3. *Whenever Algorithm 6.1 invokes the procedure* refine*, it holds that $\mathcal{X} = \{N \mid I_C^i \subseteq N \subseteq C\} \cap \texttt{Unexplored}^i = \emptyset$.*

```
while Unexplored $\neq \emptyset$ do
    $T \leftarrow$ a maximal unexplored subset of $C$
    $(sat?, E, K) \leftarrow \mathsf{isSAT}(T)$  // the model extension $E$ is not used ($T$ is unsat. or an MSS)
    if $sat?$ then
        output $T$                                    // $T$ is an MSS
        Unexplored $\leftarrow$ Unexplored$\setminus\{X \mid X \subseteq T \vee X \supseteq T\}$
        $I_C \leftarrow I \cap T$
        $I_C \leftarrow \mathsf{rotate}(T, I_C)$                       // Algorithm 6.3
    else
        $crits \leftarrow$ collect all minable critical clauses for $K$
        $K_{mus} \leftarrow \mathsf{shrink}(K, crits)$      // external black-box single MUS extractor
        Unexplored $\leftarrow$ Unexplored$\setminus\{X \mid X \supseteq K_{mus}\}$
        $I_C \leftarrow I_C \setminus K_{mus}$
        break
return $I_C$
```

Algorithm 6.2: refine(I_C)

Proof. At the first iteration of the outer loop, $\texttt{Unexplored}^i = \mathcal{P}(C)\setminus\{C\}$ and $I_C^i = C$, thus $\mathcal{X} = \emptyset$. In other iterations, the claim follows from the condition of the inner loop. □

Proposition 6.4. *Let M be an MSS of C such that $M \in \texttt{Unexplored}^i$. Then $\texttt{IMSS}_C \subseteq I_C^i \cap M \subsetneq I_C^i$.*

Proof. From the pre-condition $\mathcal{X} = \emptyset$ (Proposition 6.3), we have $I_C^i \setminus M \neq \emptyset$, thus $I_C^i \cap M \subsetneq I_C^i$. Furthermore, since both I_C^i and M are supersets of $\texttt{IMSS}_C$, we have $\texttt{IMSS}_C \subseteq I_C^i \cap M$. □

Proposition 6.5. *Let M be an MUS of C such that $M \in \texttt{Unexplored}^i$. Then $\texttt{IMSS}_C \subseteq I_C^i \setminus M \subsetneq I_C^i$.*

Proof. From the minimal hitting set duality between MUSes and MCSes, we know that for each $f \in M$, there has to exist an MCS L such that $f \in L$ and its complementing MSS $\overline{L}$ such that $f \notin \overline{L}$. Thus, we have that $\texttt{IMSS}_C \subseteq I_C^i \setminus M$. Furthermore, since $M \in \texttt{Unexplored}^i$ and from the pre-condition $\mathcal{X} = \emptyset$ (Proposition 6.3), we have $I_C^i \setminus M \neq \emptyset$, thus $I_C^i \setminus M \subsetneq I_C^i$. □

In other words, every unexplored MSS and every unexplored MUS allow us to reduce I_C. Moreover, from Observation 6.1, we know that there is at least one unexplored MSS or MUS when refine is invoked. The procedure refine is shown in Algorithm 6.2. To reduce I_C, the algorithm attempts to identify at least one unexplored MUS and possibly several unexplored MSSes. Each iteration of the algorithm starts by picking a *maximal* unexplored subset, i.e. a set $T \in \texttt{Unexplored}$ such that for each $f \in C\setminus T$ it holds that $T \cup \{f\} \notin \texttt{Unexplored}$. Then, T is checked for satisfiability via the procedure isSAT. If T is unsatisfiable, isSAT returns an unsat core K of T. Subsequently, a procedure shrink is used to find an MUS K_{mus} of K. Then, Unexplored is reduced by removing all supersets of K_{mus} from it, and I_C is reduced

```
1  $rotationQueue \leftarrow \langle N \rangle$
2  while $rotationQueue$ is not empty do
3      $M \leftarrow rotationQueue.dequeue()$
4      for $f \in C \setminus M$ do
5          for $l \in f$ do
6              $S \leftarrow (M \cup \{f\}) \setminus \{g \in M \mid \neg l \in g\}$
7              if $|M \setminus S| > threshold$ then continue
8              if $S \in$ Unexplored then
9                  $S_{mss} \leftarrow$ grow$(S)$                         // Algorithm 6.6
10                 output $S_{mss}$
11                 Unexplored $\leftarrow$ Unexplored$\setminus \{X \mid X \subseteq S_{mss} \vee X \supseteq S_{mss}\}$
12                 $I_C \leftarrow I_C \cap S_{mss}$
13                 $rotationQueue.enqueue(S_{mss})$
14 return $I_C$
```

Algorithm 6.3: rotate(N, I_C)

to $I_C \setminus K_{mus}$. In the other case, when T is satisfiable, it is guaranteed that T is an MSS of C (Observation 2.12). The algorithm reduces I_C to $I_C \cap T$, and removes all subsets and all supersets of T from Unexplored. Subsequently, T is processed by the procedure rotate that can identify additional MSSes and further reduce I_C and Unexplored. The algorithm terminates either when a first MUS is found or when Unexplored $= \emptyset$.

The procedure shrink that finds an MUS of K can be implemented via any single MUS extraction algorithm, e.g., [Bacchus and Katsirelos, 2015, Belov and Marques-Silva, 2012, Belov et al., 2014, Nadel et al., 2014]. To speed up the extraction, we provide the extraction algorithm with the set *crits* of clauses that are minable critical for K[3]. Since clauses that are critical for K has to be contained in every MUS of K, prior knowledge of such clauses is very beneficial [Liffiton et al., 2016, Bendík et al., 2018b].

[3] Note that we can possibly apply also the critical extension technique we presented in Section 5.2.4 (Algorithm 5.6) to identify additional critical clauses before the shrinking. We have not yet integrated the critical extension in our implementation of RIME, but we plan to do it.

Finally, let us explain why we allow Algorithm 6.2 to identify multiple MSSes even though finding just a single MSS would be enough to reduce I_C. The reason is that all these MSSes are identified very cheaply; we perform just a single satisfiability check per each MSS.

6.2.3 MSS Rotation

Here we describe our *MSS Rotation* technique that, based on an MSS N of C, attempts to identify additional unexplored MSSes. Moreover, it attempts to reduce the sets Unexplored and I_C. From the high-level view, MSS rotation follows the seed-grow scheme, i.e. it identifies s-seeds and grows them to MSSes. The uniqueness of MSS rotation lies in the way it finds the s-seeds. Instead of first finding an unexplored subset and then checking the unexplored subset for satisfiability, MSS rotation first finds a satisfiable subset and then checks the subset for being unexplored. Such an approach has two significant advantages. First, it is much easier to check a subset for being unexplored than to find an unexplored subset (see Section 2.3.2). Second, MSS rotation finds the satisfiable subset via a cheap deduction technique instead of employing a SAT solver. The key deduction idea is summarized in Proposition 6.6.

Proposition 6.6. *Let M be an MSS of C, f a clause such that $f \in C \backslash M$, and l a literal of f. Then, the set $S = (M \cup \{f\}) \backslash \{g \in M \mid \neg l \in g\}$ is satisfiable.*

Proof. Let π be a model of M and π_l a truth assignment that originates from π by flipping the assignment to the variable of l. Since π satisfies the whole M, then π_l satisfies at least those clauses of M that does not contain $\neg l$. Moreover, since $l \in f$ then π_l satisfies also f. □

The MSS rotation technique is shown in Algorithm 6.3. The algorithm maintains a queue *rotationQueue* of MSSes. Initially, the queue contains the input MSS N. In each iteration, the algorithm dequeues an MSS M from the queue and uses it to find new MSSes. In particular, for each $f \in C \backslash M$ and each $l \in f$, the algorithm constructs a satisfiable set S based on Proposition 6.6 and checks if S is unexplored. If S is unexplored, then S is grown to an MSS S_{mss} and the MSS is used to reduce the sets `Unexplored` and I_C. Moreover, S_{mss} is added to the queue and thus eventually used to possibly identify additional MSSes.

We employ one additional heuristic in the algorithm: instead of checking every S for being an s-seed, we skip every S such that $|M \backslash S| > threshold$ where $threshold \geqslant 1$ (line 7 in the algorithm). There are two motivations for this heuristic. First, the smaller S is, the more-likely S is a subset of some already explored MSS, i.e. the more-likely is S explored. Second, it is generally easier to grow larger s-seeds than smaller s-seeds, thus we tend to find larger s-seeds. The most suitable value of *threshold* varies for different kinds of benchmarks; thus the value of *threshold* can be specified by the user of our algorithm. In our evaluation (Section 6.4), we made an ad-hoc choice and set *threshold* to 10 for all benchmarks.

6.2.4 *Grow*

In this section, we describe the procedure `grow` that takes as an input an s-seed N and returns an unexplored MSS N_{mss} of C such that $N \subseteq N_{mss}$. The procedure is based on a simple, well-known, single MSS extraction algorithm. We first describe the base solution and then introduce several novel improvements to the base solution.

Base Solution Recall that a set N is an MSS of C if every $f \in C \backslash N$ is conflicting for N. Also, recall that if a clause f is conflicting for N, than the literals $\{\neg l \mid l \in f\}$ are backbone literals for N. The base solution is shown in Algorithm 6.4. It maintains two sets: the input set N and a set *confs* of clauses that are known to be conflicting for N. Initially, $confs = \emptyset$. In each iteration, the algorithm picks a

```
input : an unsatisfiable set $C$ of Boolean clauses
input : a satisfiable subset $N$ of $C$
output: an MSS $N_{mss}$ of $C$ such that $N \subseteq N_{mss}$
1 $confs \leftarrow \emptyset$
2 while $C \setminus (N \cup confs) \neq \emptyset$ do
3     $f \leftarrow$ pick a clause from $C \setminus (N \cup confs)$
4     $(sat?, E) \leftarrow \mathsf{isSAT}(N \cup \{f\}, \bigcup_{f \in confs}\{\neg l \mid l \in f\})$
5     if $sat?$ then $N \leftarrow E$
6     else $confs \leftarrow confs \cup \{f\}$
7 return $N$
```

Algorithm 6.4: A simple MSS extraction algorithm.

```
1 while True do
2     $E \leftarrow \mathsf{conflictExtension}(N, confs)$          // based on Definition 6.1
3     if $E = N$ then return $(N, confs)$
4     $N \leftarrow E$
5     $confs' \leftarrow$ collect all minable conflicting clauses for $N$
6     if $confs = confs'$ then return $(N, confs)$
7     $confs \leftarrow confs'$
```

Algorithm 6.5: $\mathsf{enlarge}(N, confs)$

clause $f \in C \setminus (N \cup confs)$ and checks if f is conflicting for N by checking $N \cup \{f\}$ for satisfiability via a SAT solver, denoted by isSAT. To speed up the satisfiability query, isSAT takes as an input also the set $\bigcup_{f \in confs}\{\neg l \mid l \in f\}$ of backbone literals that can be obtained from *confs* (prior knowledge of backbone literals can be very beneficial for a SAT solver). If $N \cup \{f\}$ is satisfiable, then isSAT returns a model extension E of $N \cup \{f\}$ and N is enlarged to E. In the other case, when $N \cup \{f\}$ is unsatisfiable, the clause f is added to *confs*. The computation terminates once $C \setminus (N \cup confs) = \emptyset$. The invariant of the algorithm is that N is satisfiable and all clauses in *confs* are conflicting for N. The algorithm performs up to $|C \setminus N|$ satisfiability checks (due to the use of model extensions, the number can be lower).[4]

[4] Note that Algorithm 6.4 can be seen as a domain specific variant of the domain agnostic growing algorithms we presented in Section 2.3.3 (Algorithm 2.2). The difference between Algorithm 6.4 and Algorithm 2.2 is the use of the model extension and the backbone literals to save and speed-up satisfiability checks, respectively.

Algorithm 6.4 forms a basis of several contemporary single MSS extraction algorithms (see [Marques-Silva et al., 2013a]). These algorithms extensively exploit domain specific properties of the Boolean CNF domain and are very efficient. Thus, instead of developing a custom MSS extractor, we might use as a black-box subroutine one of the existing solutions. However, the single extractors are tailored for finding only a single MSS and consequently do not (and cannot) fully exploit information that we can obtain from the overall MSS enumeration algorithm. Therefore, we propose our custom single MSS extraction algorithm that works in synergy with the overall MSS enumeration.

Key Observations Our approach is based mainly on two observations. First, we can use `Unexplored` to collect minable conflicting clauses for an s-seed N. The second observation concerns a concept of *conflict extension* that we define as follows.

Definition 6.1 (conflict extension). *Let N be an s-seed, confs a set of conflicting clauses for N, and $backs = \bigcup_{f \in confs}\{\neg l \mid l \in f\}$ the set of all backbone literals for N that can be obtained from confs. The* conflict extension *of N w.r.t. C and confs is the set $E = N \cup \{g \in C \setminus N \mid backs \cap Lits(g) \neq \emptyset\}$ where $Lits(g)$ are literals of the clause g.*

Proposition 6.7. *Let N be an s-seed, confs a set of conflicting clauses for N, and E the conflict extension of N w.r.t. C and confs. Then E is an s-seed such that $N \subseteq E$.*

Proof. Every model of N has to satisfy all the backbone literals *backs* and consequently also every clause $g \in C$ that contains at least one of the backbone literals, thus E is satisfiable. Since N is an s-seed, $E \supseteq N$ is an s-seed (Observation 2.9). □

The ability to mine conflicting clauses and the concept of conflict extension can be very powerful if we combine them. In particular, assume that we are given an s-seed N and the set *confs* of clauses that are minable conflicting for N. We can use the conflict extension to enlarge N based on *confs*. In turn, there might arise additional minable conflicting clauses for N, and so on until a fix-point is reached. We properly describe this functionality in the procedure enlarge shown in Algorithm 6.5. The input of the algorithm is an s-seed N^i and a set $confs^i$ of conflicting clauses for N^i. The output of the procedure are sets N^o, $confs^o$ such that N^o is an s-seed, $confs^o$ is a set of conflicting clauses for N^o, $N^o \supseteq N^i$, and $confs^o \supseteq confs^i$. The algorithm works iteratively. In each iteration the algorithm computes the conflict extension E of N based on *confs*, and then enlarges N to E. Subsequently, the the set $confs'$ of minable conflicting clauses for (the enlarged) N is computed, and *confs* is enlarged to $confs'$. The algorithm terminates either once the model extension does not enlarge N anymore or once we are not able to collect any new minable conflicting clauses.

Algorithm Our growing procedure is shown in Algorithm 6.6. It takes as an input an s-seed N, and it returns an unexplored MSS N_{mss} of C such that $N \subseteq N_{mss}$.

The computation starts by collecting the set *confs* of clauses that are minable conflicting for N. Subsequently, the procedure enlarge (Algorithm 6.5) is used to enlarge both *confs* and N. The main part of the algorithm is formed by a while loop. In each iteration, the algorithm picks a clause $f \in C \setminus (N \cup confs)$ and checks $N \cup \{f\}$ for satisfiability via the procedure isSAT. To make the check more efficient, we pass to isSAT the set $\bigcup_{f \in confs}\{\neg l \mid l \in f\}$ of backbone literals that can be obtained from *confs*. If $N \cup \{f\}$ is satisfiable, then we obtain its model extension E and enlarge N to E. Consequently, there might emerge additional minable conflicting clauses for N, thus we re-collect them and add them to *confs*. In the other case, when $N \cup \{f\}$ is unsatisfiable, we obtain an unsat core K of $N \cup \{f\}$ from the SAT

```
1  $confs \leftarrow$ collect all minable conflicting clauses for $N$
2  $N, confs \leftarrow \mathsf{enlarge}(N, confs)$                          // Algorithm 6.5
3  while $C \setminus (N \cup confs) \neq \emptyset$ do
4      $f \leftarrow$ pick a clause from $C \setminus (N \cup confs)$
5      $(sat?, E, K) \leftarrow \mathsf{isSAT}(N \cup \{f\}, \bigcup_{f \in confs}\{\neg l \mid l \in f\})$
6      if $sat?$ then
7          $N \leftarrow E$
8          $confs' \leftarrow$ collect all minable conflicting clauses for $N$
9          $confs \leftarrow confs \cup confs'$
10     else
11         $confs \leftarrow confs \cup \{f\}$
12         Unexplored $\leftarrow$ Unexplored$\setminus \{X \mid X \supseteq K\}$
13     $N, confs \leftarrow \mathsf{enlarge}(N, confs)$                      // Algorithm 6.5
14 return $N$
```

Algorithm 6.6: grow(N)

solver and we remove all supersets of K from `Unexplored`. Also, we add f to *confs* since f is conflicting for N. At the end of the iteration, we use the procedure enlarge to possibly further enlarge N and *confs*.

Same as in the case of the base solution (Algorithm 6.4), the invariant of the algorithm is that N is an s-seed, *confs* is a set of conflicting clauses for N, and $confs \cap N = \emptyset$. The algorithm terminates once $confs \cup N = C$, thus the final N is an unexplored MSS of C. In the worst case, the algorithm performs $|C \setminus N|$ satisfiability checks, i.e., there is no asymptotic improvement over the base solution. Yet, we have experimentally observed that the algorithm usually performs much fewer satisfiability checks and in many cases even no checks at all (see Section 6.4). This is mainly due to two reasons. First, RIME grows s-seeds that are usually very close to the resultant MSSes. Second, the procedure enlarge can often significantly enlarge both N and *confs*.

Finally, let us note that in the procedure grow, we remove from `Unexplored` only unsatisfiable sets (supersets of the unsat cores K) even though we identify also some satisfiable sets (subsets of the model extensions E). The thing is that all the model extensions E are subsets of the resultant MSS N_{mss}. We use grow in Algorithms 6.1 and 6.3 and in both cases, we remove all subsets of N_{mss} from `Unexplored` immediately after grow terminates.

6.3 *Related Work*

Whenever an MSS enumeration algorithm searches for an MSS N, it needs to deal with two tasks: (1) it has to guarantee that N is indeed an MSS, and (2) it has to guarantee that N is so far unexplored MSS. Based on these tasks, we can divide existing algorithms into two categories.

Algorithms from one category deal with the two tasks separately and are based on the seed-grow scheme. The algorithms start by finding an s-seed, i.e. with the task (1), and then they grow the s-seed into an MSS, i.e. perform the task (2). Perhaps the most famous algorithms from this category are MARCO, DAA [Bailey and Stuckey, 2005] and PDDS [Stern et al., 2012] that we already described before in Section 4.5 in the context of MUS enumeration. In particular, MARCO biased for an MSS enumeration searches for s-seeds by repeatedly picking and checking for satisfiability a minimal unexplored subset until it finds a satisfiable one, i.e., an s-seed. Since satisfiable subsets of C are naturally more concentrated among the smaller subsets, MARCO often needs only a few satisfiability checks to find the s-seed. However, the s-seeds are relatively small and thus potentially hard to grow. It is true that MARCO employs the model extension technique to enlarge the s-seed before the growing, however, there is no guarantee of how close the extension will be to an MSS. The growing itself is carried out via a black-box subroutine and can be implemented using any available single MSS extractor. As for PDDS, recall that the algorithm works in the same way as MARCO, i.e., it finds an s-seed by repeatedly checking a minimal unexplored subset for satisfiability until it identifies an s-seed. The main difference between the two algorithms is that PDDS does not exactly specify how to identify the minimal unexplored subsets, whereas authors of MARCO proposed to use the efficient technique based on the formula $map^+ \wedge map^-$. Consequently, MARCO is the one of the two that is nowadays used. As for DAA, recall that the algorithm can easily suffer from memory issues due to computing large number of minimal hitting sets simultaneously, and thus it is often practically unusable. Another algorithm from this group is our TOME which was presented in Section 4.3. However, TOME is a domain agnostic algorithm and based on our experience, it is not very efficient in the Boolean CNF domain.

Algorithms from the other group deal with the tasks (1) and (2) simultaneously. Similarly as MARCO and RIME, the algorithms use a variation of the formula $map^+ \wedge map^-$ to carry information about unexplored subsets. However, instead of using the formula separately, they combine $map^+ \wedge map^-$ with the formula C into a single Boolean formula G. Consequently, the task of checking whether a subset N of C is both satisfiable and unexplored, i.e. an s-seed, can be done via a single SAT solver query on G. To find each single MSS, the algorithms perform several such queries, and every time a new MSS is found, the formula G is modified to exclude the MSS from the further computation. To the best of our knowledge, the current state-of-the-art MSS enumerators from this category are MCSLS [Marques-Silva et al., 2013a], FLINT [Narodytska et al., 2018], and an approach from [Grégoire et al., 2018].

Finally, there have been proposed several single MSS extraction algorithms, e.g., [Felfernig et al., 2012, Bacchus et al., 2014, Grégoire et al., 2014, Mencía et al., 2015], that can be often used as a subrou-

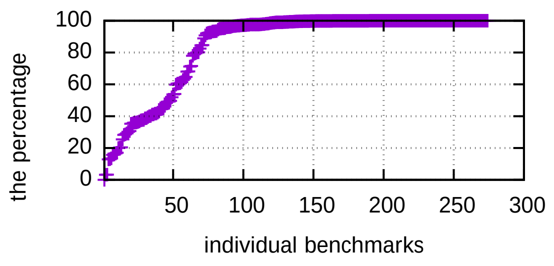


Figure 6.1: Percentage of MSSes found by MSS rotation.

tine of MSS enumeration algorithms, and several techniques [Previti et al., 2017, 2018] for caching results of SAT queries that naturally emerge during single or multiple MSS extraction.

6.4 *Experimental Evaluation*

We implemented RIME in C++ using CaDiCaL [Biere, 2018] as a SAT solver and a single MUS extractor muser2 [Belov and Marques-Silva, 2012] to perform the shrinking in Algorithm 6.2. Our tool is publicly available at:

`https://github.com/jar-ben/rime`

In this section, we experimentally compare RIME with three contemporary MSS enumeration tools: MARCO[5] [Liffiton et al., 2016], MCSLS[6] [Marques-Silva et al., 2013a], and FLINT[7] [Narodytska et al., 2018]. In case of MARCO and FLINT, the algorithms enumerate both MUSes and MSSes and can be biased either towards MUS or MSS enumeration; we used the bias for MSS enumeration.

[5] `https://sun.iwu.edu/~mliffito/marco/`

[6] The tool was kindly provided to us by its authors.

[7] The tool was kindly provided to us by its authors.

We focus on three criteria in the comparison: (1) the number of identified MSSes within a given time limit, (2) the median time to identify individual MSSes, and (3) the median number of performed satisfiability checks to identify individual MSSes. Moreover, we examine the manifestation of MSS rotation in RIME.

As benchmarks, we use the collection of 291 CNF formulae that were used in the MUS track of the SAT Competition 2011 [8] (i.e., the same benchmarks as we used in the previous two Chapters). The benchmarks range in their size from 70 to 16 million clauses and use from 26 to 4.4 million variables. All experiments were run using a time limit of 3600 seconds and computed on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux 4.19.67-2. The value of *threshold* in Algorithm 6.3 was set to 10. Complete results are available in the online appendix[9].

[8] `http://www.satcompetition.org/`

[9] `https://www.fi.muni.cz/~xbendik/phdThesis/`

6.4.1 *Manifestation of MSS Rotation*

To the best of our knowledge, MSS rotation is the very first technique that is able to identify s-seeds for the growing in polynomial time (i.e., without a SAT solver). To examine the manifestation of MSS rotation in RIME, we measured the percentage of s-seeds, and consequently MSSes, that RIME identified via the rotation technique.

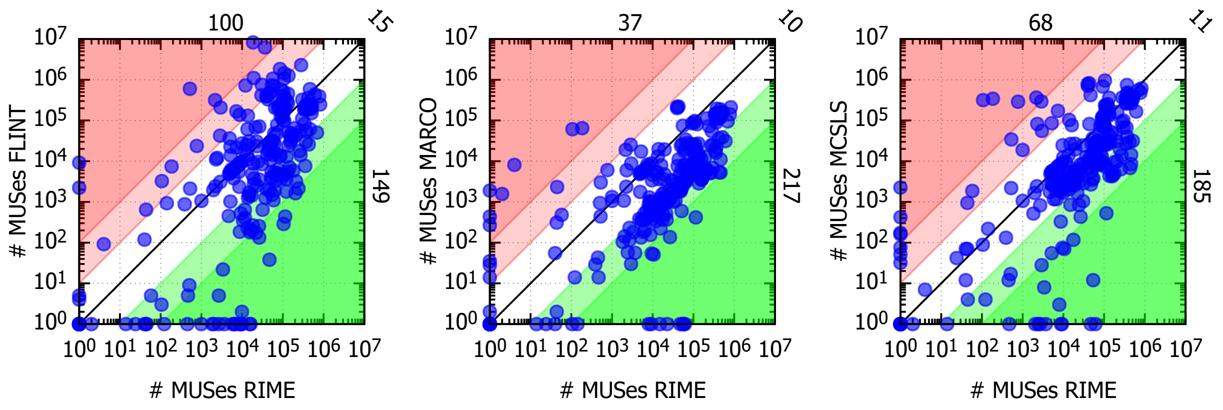


Figure 6.2: Scatter plots comparing the number of identified MSSes within first 3600 seconds.

In Figure 6.1, we show this percentage (y-axis) for individual benchmarks (x-axis); the benchmarks are sorted by the percentage. We computed the percentage only for the 273 benchmarks where RIME identified at least 5 MSSes.

You can see that the MSS rotation technique identified at least one s-seed on every benchmark. Remarkable, in the case of 203 benchmarks, the percentage is higher than 90 percent, and in the case of 155 benchmarks, the percentage is even higher than 99 percent. Hence, we conclude that MSS rotation is a crucial part of RIME.

6.4.2 *Number of Identified MSSes*

Due to possibly exponentially many MSSes in a benchmark, the complete MSS enumeration is generally intractable. Only in the case of 27 benchmarks, all the evaluated algorithms identified within the time limit all MSSes. Hence, in this section, only the remaining 264 benchmarks are the subject of our evaluation.

Figure 4.3 provides 3 scatter plots that pairwise compare RIME with its competitors on individual benchmarks. Each point in the plot shows the number of identified MSSes within the 3600 seconds by the two compared algorithms on one particular benchmark; one algorithm determines the position on the vertical axis and the other one the position on the horizontal axis. Moreover, we provide three numbers above/right/in the right corner of each plot that show the number of points above/below/on the diagonal. We also use a red and green background color to highlight different orders of magnitude (of 10). Note that the plots are in a log-scale and hence cannot show points with a zero coordinate. Therefore, we lifted the points with a zero coordinate to the 1st coordinate, i.e. the points that are exactly on the x-axis or on the y-axis show benchmarks where one of the algorithms found either only a single MSS or no MSS at all.

Besides the pair-wise comparison of the algorithms, we also provide the overall *ranking* of the algorithms on individual benchmarks (same as we did in the previous chapter). In particular, assume that for a benchmark B both RIME and FLINT found 100 MSSes, MCSLS found 80 MSSes, and MARCO found 50 MSSes. In such a case, RIME

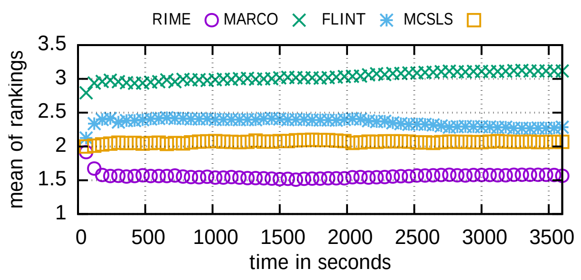


Figure 6.3: 5% truncated mean of rankings after each 60 seconds.

and FLINT share the 1st (best) rank for B , FLINT is 3rd, and MARCO is on the 4th position. For each of the algorithms, we computed the 5 percent truncated arithmetic mean of the rankings on all benchmarks (i.e., for each algorithm, we discarded the 5 percent of benchmarks where the algorithm achieved the best and the worst ranking). To capture the change of the performance of the algorithms in time, we computed the truncated mean of the rankings after each subsequent 60 seconds of the computation. The results are shown in Figure 6.3.

We conclude that RIME conclusively dominated all of its competitors. Based on the scatter plots, the tightest competitor of RIME is FLINT which was able to outperform RIME on 100 benchmarks, but it lost to RIME on 149 benchmarks. RIME also steadily maintained the best (truncated mean) ranking with the value around 1.56 during the whole time period. The second-best ranking was achieved by MCSLS which steadily maintained a ranking around 2.1. FLINT maintained during the first 2000 seconds a ranking around 2.4 and then it gradually improved its ranking towards the final value 2.3. Finally, MARCO maintained the worst ranking with the final value 3.1

The fact that MCSLS ranks on average better than FLINT can be surprising considering the pair-wise scatter plot comparison where FLINT performed better against RIME than MCSLS. Based on our closer examination of the experimental results, it is the case that FLINT performs very well on many benchmarks (i.e., ranks the best), however, there are also many benchmarks where FLINT performs poorly (i.e. ranks the worst or the second-worst). On the other hand, MCSLS achieved the worst ranking only on a few benchmarks.

6.4.3 Time and Checks per MSS

Here, we examine the number of elapsed seconds and the number of performed satisfiability checks to identify each subsequent MSS. These numbers naturally differ for individual benchmarks, thus we focus on median values. The plot in Figure 6.4 shows the median number of elapsed seconds required to identify the first 5000 MSSes. A point with coordinates (x, y) states that the median of elapsed seconds to output the first x MSSes is y. We used only 91 benchmarks to compute the medians since only in those benchmarks all the algorithms found at least 5000 MSSes. The figure shows that all the algorithms, except FLINT, produce individual MSSes at a rela-

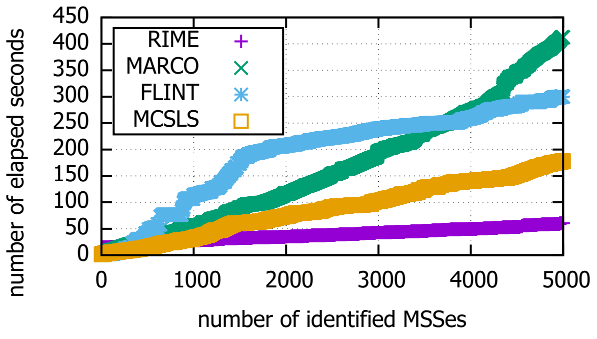


Figure 6.4: Median number of elapsed seconds per MSS to output the first 5000 MSSes.

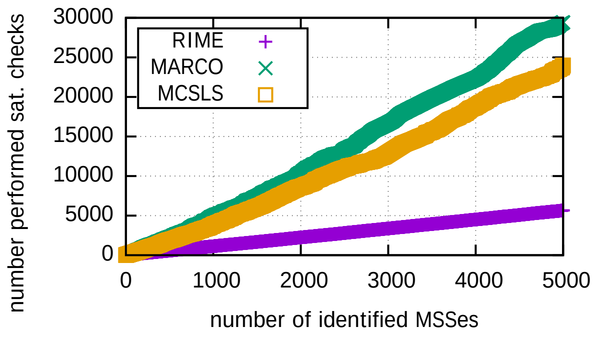


Figure 6.5: Median number of performed satisfiability checks per MSS to output the first 5000 MSSes.

tively steady rate. FLINT was rather slow in identifying the first 1500 MSSes, however, then it speeded up. The median amount of elapsed time to find the 5000th MSS by RIME, MCSLS, FLINT, and MARCO was 60, 178, 300, and 412 seconds, respectively.

MSS enumeration naturally subsumes checking subsets of the input Boolean formula for satisfiability. Based on our experience, performing these checks is the most expensive part of an MSS enumeration algorithm. In Figure 6.5, we show the median number of satisfiability checks performed to identify the first 5000 MSSes (computed from the 91 benchmarks). Unfortunately, we were unable to measure the number of checks for FLINT since its authors provided us only with a binary version of the tool that does not provide this information. We can see that the median number of performed checks to output the 5000th MSS by RIME, MCSLS, and MARCO was 5656, 24002, and 29432, respectively. Thus, to find each single MSS, RIME, MCSLS, and MARCO performed on average around 1.13, 4.8, and 5.9 satisfiability checks.

Remarkably, in the case of 55 out of all 291 benchmarks, RIME performed fewer satisfiability checks than what was the number of identified MSSes. This was achieved mainly due to our novel techniques of MSS rotation, which allowed us to find s-seeds without a SAT solver, and conflict extension which often allowed us to grow an s-seed without performing even a single satisfiability check.

We conclude that it is the frugality of RIME w.r.t. the number of performed satisfiability checks what allowed it to so substantially outperform all of its competitors. RIME found significantly more MSSes than its competitors on a majority of benchmarks; the difference was often several orders of magnitude. Moreover, in terms of the median values, RIME is three times faster than its closest competitor MCSLS, five times faster than FLINT, and seven times faster than MARCO.

Part III

MUS and MSS/MCS Counting in the Boolean CNF Domain

7
Boolean CNF MUS Counting

Same as in the previous two chapters, here our goal is to analyze a set C of Boolean clauses and the $\mathbf{P}_1$-minimal and $\mathbf{P}_0$-maximal subsets are the minimal unsatisfiable and maximal satisfiable subsets of C, respectively. However, instead of MUS or MSS/MCS enumeration as we did in the previous two chapters, here we present an algorithm for counting the number of MUSes of C, called AMUSIC [Bendík and Meel, 2020].

Although the research on MUSes has been pioneered already in 1987 by Reiter [Reiter, 1987], the early techniques for MUS identification were quite inefficient. Consequently, the first applications of MUSes primarily required identification of just a single MUS of the input formula. With an improvement of the scalability of MUS enumeration techniques about a decade and a half ago [Bailey and Stuckey, 2005], new applications that require identification of multiple or even all MUSes arised. In the past few years, MUS identification/enumeration approaches improved even more, and thus researchers now have sought to investigate other extensions of MUSes and their applications. One such application is MUS counting, i.e., the problem is to count the number of MUSes of an input unsatisfiable CNF formula C. Current applications of MUS counting include mainly various inconsistency metrics for general propositional knowledge bases such as those presented in [Hunter and Konieczny, 2008, Thimm, 2018, Mu, 2019].

In contrast to the progress in the design of efficient MUS identification techniques, the work on MUS counting is still in its nascent stages. The current approach for MUS counting is the most straightforward one, i.e., to employ a complete MUS enumeration algorithm, e.g., [Stern et al., 2012, Liffiton and Malik, 2013, Bacchus and Katsirelos, 2016, Bendík and Černá, 2020c], to explicitly identify all MUSes. However, as we noted in Section 2, there can be in general up to exponentially many MUSes w.r.t. the number of clauses in C. Thus the complete MUS enumeration is still often practically intractable (which we in fact witnessed in our experimental evaluation in Section 5.4). Thus, MUS enumeration algorithms cannot be used for MUS counting when the complete enumeration is intractable. In this context, one asks the question: *whether it is possible to count the MUSes without performing an explicit MUS enumeration?*

The primary contribution of this chapter is an affirmative answer to the above question. In particular, we present a probabilistic counter, called AMUSIC, that takes in a CNF formula C, a tolerance parameter ε, a confidence parameter δ, and it returns an estimate guaranteed to be within $(1+\varepsilon)$-multiplicative factor of the exact count with confidence at least $1-\delta$. Crucially, for C defined over n clauses, AMUSIC explicitly identifies only $\mathcal{O}(\log n \cdot \log(1/\delta) \cdot (\varepsilon)^{-2})$ many MUSes even though the number of MUSes can be exponential in n.

The design of AMUSIC is inspired by recent successes in the design of efficient XOR hashing-based techniques [Chakraborty et al., 2013, 2016] for the problem of model counting, i.e., given a Boolean formula G, compute the number of models of G. We observe that both the problems are defined over a power-set structure. In MUS counting, the goal is to count MUSes in the power-set of C, whereas in model counting, the goal is to count models in the power-set that represents all valuations of variables of G. A few years ago, there was proposed an algorithm, called ApproxMC [Chakraborty et al., 2016, Soos and Meel, 2019], for approximate model counting that also provides the (ϵ, δ) guarantees. ApproxMC is currently already in its fourth version, ApproxMC4 [Soos et al., 2020]. The base idea of ApproxMC4 is to partition the power-set into *nCells* small cells, then pick one of the cells, and count the number *inCell* of models in the cell. The total model count is then estimated as *nCells* $\times$ *inCell*. Our algorithm for MUS counting is based on ApproxMC4. We adopt the high-level idea to partition the power-set of C into small cells and then estimate the total MUS count based on an MUS count in a single cell. The difference between ApproxMC4 and AMUSIC lies in the way of counting the target elements (models vs. MUSes) in a single cell; we propose novel MUS specific techniques to deal with this task. In particular, our contribution is the following:

- We introduce a QBF (quantified Boolean formula) encoding for the problem of counting MUSes in a single cell and use a Σ_3^P oracle (a 3QBF solver) to solve it.

- Let $\texttt{UMUS}_C$ and $\texttt{IMUS}_C$ be the union and the intersection of all MUSes of C, respectively. We observe that every MUS of C (1) contains $\texttt{IMUS}_C$ and (2) is contained in $\texttt{UMUS}_C$. Consequently, if we determine the sets $\texttt{UMUS}_C$ and $\texttt{IMUS}_C$, then we can significantly speed up the identification of MUSes in a cell.

- We propose a novel approaches for computing the union $\texttt{UMUS}_C$ and the intersection $\texttt{IMUS}_C$ of all MUSes of C.

- We implement AMUSIC and conduct an extensive empirical evaluation on a set of *scalable* benchmarks. We observe that AMUSIC is able to compute estimates for problems clearly beyond the reach of existing enumeration-based techniques. We experimentally evaluate the *accuracy* of AMUSIC. In particular, we observe that the estimates computed by AMUSIC are significantly closer to true count than the theoretical guarantees provided by AMUSIC.

Our work opens up several new interesting avenues of research. From a theoretical perspective, we make polynomially many calls to a Σ_3^P oracle while the problem of finding an MUS is known to be in FP^{NP}, i.e. an MUS can be found in polynomial time by executing a polynomial number of calls to an NP-oracle [Chen and Toda, 1995, Marques-Silva and Janota, 2014]. Contrasting this to model counting techniques, where approximate counter makes polynomially many calls to an NP-oracle when the underlying problem of finding satisfying assignment is NP-complete, a natural question is to close the gap and seek to design an MUS counting algorithm with polynomially many invocations of an FP^{NP} oracle. From a practitioner perspective, our work calls for a design of MUS techniques with native support for XORs; the pursuit of native support for XOR in the context of SAT solvers have led to an exciting line of work over the past decade [Soos et al., 2009, Soos and Meel, 2019].

7.1 Preliminaries and Problem Formulation

Throughout the whole chapter, we use C to denote the input unsatisfiable set of Boolean clauses, i.e. a CNF formula. We use $\mathtt{AMUS}_C$ to denote the set of all MUSes of C. Furthermore, we write $\mathtt{UMUS}_C$ and $\mathtt{IMUS}_C$ to denote the union and the intersection of all MUSes of C, respectively. Note that every subset S of C can be expressed as a bit-vector over the alphabet $\{0,1\}$; for example, if $C = \{c_1, c_2, c_3, c_4\}$ and $S = \{c_1, c_4\}$, then the bit-vector representation of S is 1001. As in the previous two chapters, given an unsatisfiable subset N of C and a clause $c \in N$, we say that c is *critical* for N iff $N \setminus \{c\}$ is satisfiable.

A *quantified Boolean formula*, shortly *QBF*, is a Boolean formula where each variable is either universally ($\forall$) or existentially ($\exists$) quantified. We write $Q_1 \cdots Q_k$-QBF, where $Q_1, \dots Q_k \in \{\forall, \exists\}$, to denote the class of QBF with a particular type of alternation of the quantifiers, e.g., $\exists\forall$-QBF or $\exists\forall\exists$-QBF. Every QBF is either true (valid) or false (invalid). The problem of deciding validity of a formula in $Q_1 \cdots Q_k$-QBF where $Q_1 = \exists$ is Σ_k^P-complete [Meyer and Stockmeyer, 1972].

When it is clear from the context, we write just *formula* to denote either a QBF or a Boolean formula in CNF. **Importantly**, due to technical reasons, we use $\{1, 0\}$ as the Boolean values instead of $\{True, False\}$ as we did in the other chapters.

We write $Pr[O : \mathbb{P}]$ to denote the probability of an outcome O when sampling from a probability space $\mathbb{P}$. When $\mathbb{P}$ is clear from the context, we write just $Pr[O]$.

Hash Functions

Let n and m be positive integers such that $m < n$. By $\{1,0\}^n$ we denote the set of all bit-vectors of length n over the alphabet $\{1,0\}$.

Given a vector $v \in \{1,0\}^n$ and $i \in \{1,\dots,n\}$, we write $v[i]$ to denote the i-th bit of v. A hash function h from a family $H_{xor}(n,m)$ of hash functions maps $\{1,0\}^n$ to $\{1,0\}^m$. The family $H_{xor}(n,m)$ is defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n}(a_{i,k} \wedge y[k]))\, \mathit{for\,all}\, 1 \leqslant i \leqslant m\}$, where $\oplus$ and $\wedge$ denote the Boolean XOR and AND operators, respectively, and $a_{i,k} \in \{1,0\}$ for all $1 \leqslant i \leqslant m$ and $1 \leqslant k \leqslant n$.

To choose a hash function uniformly at random from $H_{xor}(n,m)$, we randomly and independently choose the values of $a_{i,k}$. It has been shown [Gomes et al., 2006] that the family $H_{xor}(n,m)$ is pairwise independent, also known as strongly 2-universal. In particular, let us by $h \leftarrow H_{xor}(n,m)$ denote the probability space obtained by choosing a hash function h uniformly at random from $H_{xor}(n,m)$. The property of pairwise independence guarantees that for all $\alpha_1, \alpha_2 \in \{1,0\}^m$ and for all distinct $y_1, y_2 \in \{1,0\}^n$, $Pr[\bigwedge_{i=1}^{2} h(y_i) = \alpha_i : h \leftarrow H_{xor}(n,m)] = 2^{-2m}$.

We say that a hash function $h \in H_{xor}(n,m)$ *partitions* $\{0,1\}^n$ into 2^m *cells*. Furthermore, given a hash function $h \in H_{xor}(n,m)$ and a cell $\alpha \in \{1,0\}^m$ of h, we define their *prefix-slices*. In particular, for every $k \in \{1,\dots,m\}$, the k^{th} *prefix* of h, denoted $h^{(k)}$, is a map from $\{1,0\}^n$ to $\{1,0\}^k$ such that $h^{(k)}(y)[i] = h(y)[i]$ for all $y \in \{1,0\}^n$ and for all $i \in \{1,\dots,k\}$. Similarly, the k^{th} prefix of α, denoted $\alpha^{(k)}$, is an element of $\{1,0\}^k$ such that $\alpha^{(k)}[i] = \alpha[i]$ for all $i \in \{1,\dots,k\}$. Intuitively, a cell $\alpha^{(k)}$ of $h^{(k)}$ originates by merging the two cells of $h^{(k+1)}$ that differ only in the last bit.

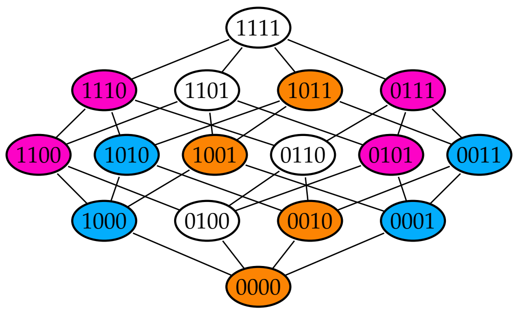
In our work, we use hash functions from the family $H_{xor}(n,m)$ to partition the power-set $\mathcal{P}(C)$ of the given Boolean formula C into 2^m cells. In particular, each subset N of C is uniquely identified by its bit-vector representation $bit(N) \in \{1,0\}^{|C|}$, i.e., a hash function $h \in H_{xor}(|C|,m)$ divides the power-set $\mathcal{P}(C)$ into 2^m cells. Furthermore, given a cell $\alpha \in \{0,1\}^m$, let us by $\mathtt{AMU}_{\langle C,h,\alpha\rangle}$ denote the set of all MUSes in the cell α of h; formally, $\mathtt{AMU}_{\langle C,h,\alpha\rangle} = \{M \in \mathtt{AMUS}_C \mid h(bit(M)) = \alpha\}$, where $bit(M)$ is the bit-vector representation of M. The following observation is crucial for our work.

Proposition 7.1. *For every formula C, $m \in \{1,\dots,|C|-1\}$, a hash function $h \in H_{xor}(|C|,m)$, and a cell $\alpha \in \{0,1\}^m$ it holds that: $\mathtt{AMU}_{\langle C,h^{(i)},\alpha^{(i)}\rangle} \supseteq \mathtt{AMU}_{\langle C,h^{(j)},\alpha^{(j)}\rangle}$ for every $i < j$.*

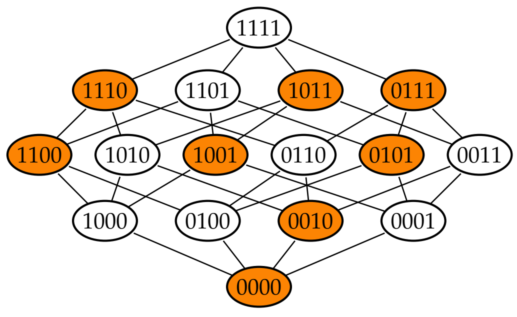
Proof. Let M be an MUS from $\mathtt{AMU}_{\langle C,h^{(j)},\alpha^{(j)}\rangle}$, i.e., $h^{(j)}(bit(M)) = \alpha^{(j)}$. Since $i < j$, then by the definition of the prefix slices, $\alpha^{(i)}[k] = \alpha^{(j)}[k]$ and $h^{(i)}(bit(M))[k] = h^{(j)}(bit(M))[k]$ for all $k \in \{1,\dots,i\}$. Consequently, $h^{(i)}(bit(M)) = \alpha^{(i)}$, i.e., M is in $\mathtt{AMU}_{\langle C,h^{(i)},\alpha^{(i)}\rangle}$. Since M is arbitrary, we conclude that $\mathtt{AMU}_{\langle C,h^{(i)},\alpha^{(i)}\rangle} \supseteq \mathtt{AMU}_{\langle C,h^{(j)},\alpha^{(j)}\rangle}$. □

Example 7.1. *Assume that we are given a formula C such that $|C| = 4$ and a hash function $h \in H_{xor}(4,2)$ that is defined via the following values of individual $a_{i,k}$:*

$$a_{1,0} = 0,\ \ a_{1,1} = 1,\ \ a_{1,2} = 1,\ \ a_{1,3} = 0,\ \ a_{1,4} = 1$$
$$a_{2,0} = 0,\ \ a_{2,1} = 1,\ \ a_{2,2} = 0,\ \ a_{2,3} = 0,\ \ a_{2,4} = 1$$



(a)



(b)

Figure 7.1: Illustration of the partition of $\mathcal{P}(C)$ by $h = h^{(2)}$ and $h^{(1)}$ from Example 7.1. Figure (a) illustrates $h^{(2)}$ with 4 cells: $\alpha_1 = 00$, $\alpha_2 = 01$, $\alpha_3 = 10$, and $\alpha_4 = 11$. Figure (b) illustrates $h^{(1)}$ with 2 cells: $\alpha_1 = 0$ and $\alpha_2 = 1$.

The hash function partitions $\mathcal{P}(C)$ into 4 cells. For example, $h(1100) = 01$ since $h(1100)[1] = 0 \oplus (1 \wedge 1) \oplus (1 \wedge 1) \oplus (0 \wedge 0) \oplus (1 \wedge 0) = 0$ and $h(1100)[2] = 0 \oplus (1 \wedge 1) \oplus (0 \wedge 1) \oplus (0 \wedge 0) \oplus (1 \wedge 0) = 1$. Figure 7.1 illustrates the whole partition and also illustrates the partition given by the prefix $h^{(1)}$ of h.

7.1.1 *Problem Definitions*

In this part of the thesis, we are concerned with the following problems.

Name: (ϵ, δ)-`#MUS` problem
Input: A formula C, a tolerance $\epsilon > 0$, and a confidence $1 - \delta \in (0, 1]$.
Output: A number c such that $Pr[|\texttt{AMUS}_C|/(1+\epsilon) \leqslant c \leqslant |\texttt{AMUS}_C| \cdot (1+\epsilon)] \geqslant 1 - \delta$.

Name: `MUS-membership` problem
Input: A formula C and a clause $f \in C$.
Output: *True* if there is an MUS $M \in \texttt{AMUS}_C$ such that $f \in M$ and *False* otherwise.

Name: `MUS-union` problem
Input: A formula C.
Output: The union $\texttt{UMUS}_C$ of all MUSes of C.

Name: `MUS-intersection` problem
Input: A formula C.
Output: The intersection $\texttt{IMUS}_C$ of all MUSes of C.

Name: (ϵ, δ)-`#SAT` problem
Input: A formula C, a tolerance $\epsilon > 0$, and a confidence $1 - \delta \in (0, 1]$.
Output: A number m such that $Pr[m/(1+\epsilon) \leqslant c \leqslant m \cdot (1+\epsilon)] \geqslant 1 - \delta$, where m is the number of models of C.

Our main goal is to provide a solution to the (ϵ, δ)-`#MUS` problem. We also deal with the `MUS-membership`, `MUS-union` and `MUS-intersection` problems since these problems emerge in our approach for solving the (ϵ, δ)-`#MUS` problem. Finally, we do not focus on solving the (ϵ, δ)-`#SAT` problem, however the problem is closely related to the (ϵ, δ)-`#MUS` problem.

7.2 *Related Work*

It is well-known (see e.g., [de Kleer and Williams, 1987, Reiter, 1987, Liffiton and Sakallah, 2008]) that a clause $f \in C$ belongs to $\texttt{IMUS}_C$ iff f is critical for C. Therefore, to compute $\texttt{IMUS}_C$, one can simply check each $f \in C$ for being critical for C. We are not aware of any work that has focused on the `MUS-intersection` problem in more detail.

The `MUS-union` problem was recently investigated by Mencia et al. [Mencía et al., 2019]. Their algorithm is based on gradually refining an *under*-approximation of $\texttt{UMUS}_C$ until the exact $\texttt{UMUS}_C$ is computed. Unfortunately, the authors experimentally show that their algorithm often fails to find the exact $\texttt{UMUS}_C$ within a reasonable time limit even for relatively small input instances (only an under-approximation is computed). In this chapter, we propose an approach that works in the other way: we initially compute an *over-approximation* of $\texttt{UMUS}_C$ and then gradually refine the approximation to eventually get $\texttt{UMUS}_C$. Another related research was conducted by Janota and Marques-Silva [Janota and Marques-Silva, 2011] who proposed several QBF encodings for solving the `MUS-membership` problem. Although they did not focus on finding $\texttt{UMUS}_C$, one can identify $\texttt{UMUS}_C$ by solving the `MUS-membership` problem for each $c \in C$.

As for counting the number of MUSes of C, we are not aware of any previous work dedicated to this problem. Yet, there have been proposed plenty of algorithms and tools for enumerating/identifying all MUSes of C (please refer to Section 5.3 for an overview). Clearly, if we enumerate all MUSes of C, then we obtain the exact value of $|\texttt{AMUS}_C|$, and thus we also solve the (ϵ, δ)-`#MUS` problem. However, since there can be up to exponentially many of MUSes w.r.t. $|C|$, MUS enumeration algorithms are often not able to complete the enumeration in a reasonable time and thus are not able to find the value of $|\texttt{AMUS}_C|$.

Very similar to the (ϵ, δ)-`#MUS` problem is the (ϵ, δ)-`#SAT` problem. Both problems involve the same probabilistic and approximation guarantees. Moreover, both problems are defined over a power-set structure. In MUS counting, the goal is to count MUSes in $\mathcal{P}(C)$, whereas in model counting, the goal is to count models in $\mathcal{P}(\mathit{Vars}(C))$. In this chapter, we propose an algorithm for solving the (ϵ, δ)-`#MUS` problem that is based on the approximate model counter ApproxMC4 [Chakraborty et al., 2013, 2016, Soos et al., 2020]. In particular, we keep the high-level idea of ApproxMC4 for processing/exploring the power-set structure, and we propose new low-level techniques that are specific for MUS counting.

7.3 AMUSIC*: A Hashing-Based MUS Counter*

We now describe AMUSIC, a hashing-based algorithm designed to solve the (ϵ, δ)-#MUS problem. The name of the algorithm is an acronym for Approximate Minimal Unsatisfiable Subsets Implicit Counter[1]. AMUSIC is based on ApproxMC4, which is a hashing-based

[1] Note that the name of the algorithm also captures the musical qualities of its authors.

algorithm to solve the (ϵ, δ)-#SAT problem. As such, while the high-level structure of AMUSIC and ApproxMC4 share close similarities, the two algorithms differ significantly in the design of core technical subroutines.

We first discuss the high-level structure of AMUSIC in Section 7.3.1. We then present the key technical contributions of this chapter: the design of core subroutines of AMUSIC in Sections 7.3.3, 7.3.4 and 7.3.5.

7.3.1 *Algorithmic Overview*

The main procedure of AMUSIC is presented in Algorithm 7.1. The algorithm takes as an input a Boolean formula C in CNF, a tolerance parameter ϵ (> 0), and a confidence parameter $\delta \in (0, 1]$, and returns an estimate of $|\texttt{AMUS}_C|$ within tolerance ϵ and with confidence at least $1 - \delta$. Similar to ApproxMC4, we first check whether $|\texttt{AMUS}_C|$ is smaller than a specific threshold that is a function of ϵ. This check is carried out via an MUS enumeration algorithm, denoted findMUSes, that returns a set Y of MUSes of C such that $|Y| = \texttt{min}(\mathsf{threshold}, |\texttt{AMUS}_C|)$. If $|Y| < \mathsf{threshold}$, the algorithm terminates while identifying the exact value of $|\texttt{AMUS}_C|$. In a significant departure from ApproxMC4, AMUSIC subsequently computes the union ($\texttt{UMUS}_C$) and the intersection ($\texttt{IMUS}_C$) of all MUSes of C by invoking the subroutines getUMU and getIMU, respectively. Through the lens of set representation of the CNF formulas, we can view $\texttt{UMUS}_C$ as another CNF formula, G. Our key observation is that $\texttt{AMUS}_C = \texttt{AMUS}_G$ (see Section 7.3.2), thus instead of working with the whole C, we can focus only on G. The rest of the main procedure is similar to ApproxMC4, i.e., we repeatedly invoke the core subroutine called AMUSICCore. The subroutine attempts to find an estimate c of $|\texttt{AMUS}_G|$ within the tolerance ϵ. Briefly, to find the estimate, the subroutine partitions $\mathcal{P}(G)$ into nCells cells, then picks one of the cells, and counts the number nSols of MUSes in the cell. The pair $(\mathsf{nCells}, \mathsf{nSols})$ is returned by AMUSICCore, and the estimate c of $|\texttt{AMUS}_G|$ is then computed as $\mathsf{nSols} \times \mathsf{nCells}$. There is a small chance that AMUSICCore fails to find the estimate; in such a case $\mathsf{nCells} = \mathsf{nSols} = \texttt{null}$. Individual estimates are stored in a list S. After the final invocation of AMUSICCore, AMUSIC computes the median of the list S and returns the median as the final estimate of $|\texttt{AMUS}_G|$. The total number of invocations of AMUSICCore is in $\mathcal{O}(\log(1/\delta))$ which is enough to ensure the required confidence $1 - \delta$ (details on assurance of the (ϵ, δ) guarantees are provided in Section 7.3.2).

We now turn to AMUSICCore which is described in Algorithm 7.2. The partition of $\mathcal{P}(G)$ into nCells cells is made via a hash function h from $H_{xor}(|G|, m)$, i.e. $\mathsf{nCells} = 2^m$. The choice of m is a crucial part of the algorithm as it regulates the size of the cells. Intuitively, it is easier to identify all MUSes of a small cell. However, on the contrary, the use of small cells does not allow to achieve a reasonable tolerance. For example, based on a cell that contains two subsets of

```
input : an unsatisfiable set $C$ of Boolean clauses
input : a tolerance $\epsilon > 0$
input : a confidence parameter $\delta$
output: a number $c$ such that $Pr[|\texttt{AMUS}_C|/(1+\epsilon) \leqslant c \leqslant |\texttt{AMUS}_C| \cdot (1+\epsilon)] \geqslant 1-\delta$
1  threshold $\leftarrow 1 + 9.84(1 + \frac{\epsilon}{1+\epsilon})(1 + \frac{1}{\epsilon})^2$
2  $Y \leftarrow$ findMUSes($C$, threshold)              // external MUS enumeration algorithm
3  if $|Y| <$ threshold then return $|Y|$
4  $G \leftarrow$ getUMU($C$)                            // Algorithm 7.5
5  $\mathrm{I}_G \leftarrow$ getIMU($G$)                         // Algorithm 7.6
6  nCells $\leftarrow 2$; $S \leftarrow$ emptyList; $iter \leftarrow 0$
7  while $iter < \lceil 17 \log_2(3/\delta) \rceil$ do
8      $iter \leftarrow iter + 1$
9      (nCells, nSols) $\leftarrow$ AMUSICCore($G, \mathrm{I}_G$, threshold, nCells)   // Algorithm 7.2
10     if nCells $\neq$ null then addToList($S$, nCells $\times$ nSols)
11 return findMedian($S$)
```

Algorithm 7.1: Probabilistic approximate MUS counting algorithm AMUSIC.

```
1  Choose $h$ at random from $H_{xor}(|G|, |G|-1)$
2  Choose $\alpha$ at random from $\{0,1\}^{|G|-1}$
3  nSols $\leftarrow$ countInCell($G, \mathrm{I}_G, h, \alpha$, threshold)          // Algorithm 7.4
4  if nSols = threshold then return (null, null)
5  if $prevNCells \neq$ null then $mPrev \leftarrow \log_2 prevNCells$
6  else $mPrev \leftarrow 2$
7  (nCells, nSols) $\leftarrow$ logMUSSearch($G, \mathrm{I}_G, h, \alpha$, threshold, $mPrev$)   // Algorithm 7.3
8  return (nCells, nSols)
```

Algorithm 7.2: AMUSICCore(G, I_G, threshold, $prevNCells$)

G we can get only three possible estimates on the MUS count, since there can be only zero, one, or two MUSes in the cell. Based on ApproxMC4, we choose m such that a cell given by a hash function $h \in H_{xor}(|G|, m)$ contains *almost* threshold many MUSes.

In particular, the computation of AMUSICCore starts by choosing at random a hash function h from $H_{xor}(|G|, |G|-1)$ and a cell α at random from $\{0,1\}^{|G|-1}$. Subsequently, the algorithm tends to identify m^{th} prefixes $h^{(m)}$ and $\alpha^{(m)}$ of h and α, respectively, such that $|\texttt{AMU}_{\langle G, h^{(m)}, \alpha^{(m)} \rangle}| <$ threshold and $|\texttt{AMU}_{\langle G, h^{(m-1)}, \alpha^{(m-1)} \rangle}| \geqslant$ threshold. Recall that $\texttt{AMU}_{\langle G, h^{(1)}, \alpha^{(1)} \rangle} \supseteq \cdots \supseteq \texttt{AMU}_{\langle G, h^{(|G|-1)}, \alpha^{(|G|-1)} \rangle}$ (Proposition 7.1, Section 7.1). We also know that the cell $\alpha^{(0)}$, i.e. the whole $\mathcal{P}(G)$, contains at least threshold MUSes (see Algorithm 7.1, line 3). Consequently, there can exist at most one such m, and it exists if and only if $|\texttt{AMU}_{\langle G, h^{(|G|-1)}, \alpha^{(|G|-1)} \rangle}| <$ threshold. Therefore, the algorithm first checks whether $|\texttt{AMU}_{\langle G, h^{(|G|-1)}, \alpha^{(|G|-1)} \rangle}| <$ threshold. The check is carried via a procedure countInCell that returns the number nSols $= \texttt{min}(|\texttt{AMU}_{\langle G, h^{(|G|-1)}, \alpha^{(|G|-1)} \rangle}|, \text{threshold})$. If nSols = threshold, then the procedure AMUSICCore fails to find the estimate of $|\texttt{AMUS}_G|$ and terminates. Otherwise, a procedure logMUSSearch is used to find the

```
$low \leftarrow 0$; $high \leftarrow |G| - 1$
$m \leftarrow mPrev$
$finalCount \leftarrow$ null
$count \leftarrow$ CountInCell($G, I_G, h^{(m)}, \alpha^{(m)}$, threshold)        // Algorithm 7.4
if $count =$ threshold then
    $low \leftarrow m$
else
    $finalCount \leftarrow count$
    $count \leftarrow$ CountInCell($G, I_G, h^{(m-1)}, \alpha^{(m-1)}$, threshold)        // Algorithm 7.4
    if $count =$ threshold then
        return $(2^m, finalCount)$
    else
        $high \leftarrow m - 1$
        $finalCount \leftarrow count$
while $high - low > 1$ do
    $m \leftarrow \lceil (low + high)/2 \rceil$
    $count \leftarrow$ CountInCell($G, I_G, h^{(m)}, \alpha^{(m)}$, threshold)        // Algorithm 7.4
    if $count =$ threshold then
        $low \leftarrow m$
    else
        $high \leftarrow m$
        $finalCount \leftarrow count$
return $(2^{high}, finalCount)$
```

Algorithm 7.3: logMUSSearch(G, I_G, h, α, threshold, $mPrev$)

required value of m together with the number nSols of MUSes in $\alpha^{(m)}$.

The procedure logMUSSearch, shown in Algorithm 7.3, is built from two ingredients. First, based on our empirical experience, the target value of m is often similar for repeated calls of AMUSICCore. Therefore, AMUSIC keeps the value $mPrev$ of m from previous iteration and in each call of logMUSSearch, we start by testing whether $mPrev$ is again the target value. In particular, logMUSSearch employs the procedure countInCell to find out whether the cells $\alpha^{(mPrev)}$ and $\alpha^{(mPrev-1)}$ contain fewer and at least as many MUSes as threshold, respectively (lines 4 and 9). Recall that countInCell returns either the exact MUS count in a given cell if the count is lower than threshold, and threshold otherwise. If it is the case that $\alpha^{(mPrev)}$ contains fewer MUSes than threshold and $\alpha^{(mPrev-1)}$ contains more MUSes than threshold, then logMUSSearch terminates and returns the number 2^{mPrev} of cells constituted by $h^{(mPrev)}$ together with the number of MUSes in the cell $\alpha^{(mPrev)}$.

If $mPrev$ is not the target value, the second ingredient comes into play. Since it holds that $\texttt{AMU}_{\langle G, h^{(1)}, \alpha^{(1)} \rangle} \supseteq \cdots \supseteq \texttt{AMU}_{\langle G, h^{(|G|-1)}, \alpha^{(|G|-1)} \rangle}$, i.e. the prefix slices are totally ordered w.r.t. the MUS count, we can find the target value of m via binary search. In the pseudocode, we use the variables low and $high$ to maintain the searching interval where the binary search operates. The invariant of the search is that $|\mathtt{AMU}_{\langle G, h^{(low)}, \alpha^{(low)}\rangle}| \geqslant \mathsf{threshold}$ and $|\mathtt{AMU}_{\langle G, h^{(high)}, \alpha^{(high)}\rangle}| < \mathsf{threshold}$. Thus, once $high - low = 1$, it is guaranteed that *high* is the target value of m. The return value is the number of cells of $h^{(high)}$, i.e., 2^{high}, and the number of MUSes in the cell $\alpha^{(high)}$ (we use the variable *finalCount* to maintain this MUS count). The initial values of *low* and *high* are set based on the two initial calls of countInCell (the calls on lines 4 and 9). In particular, the initial search interval is either $[0, \dots, mPrev - 1]$ or $[mPrev, \dots, |G| - 1]$.

Finally, let us note that in ApproxMC4, the procedure countInCell is called BSAT and it is implemented via an NP oracle, whereas we use a Σ_3^P oracle to implement the procedure (see Section 7.3.3). The high-level functionality is the same: the procedures use up to threshold calls of the oracle to check whether the number of the target elements (models vs. MUSes) in a cell is lower than threshold.

7.3.2 *Analysis and Comparison With* ApproxMC4

Following from the discussion above, there are three crucial technical differences between AMUSIC and ApproxMC4: (1) the implementation of the subroutine countInCell in the context of MUSes, (2) computation of the intersection $\mathtt{IMUS}_C$ of all MUSes of C and its usage in countInCell, and (3) computation of the union $\mathtt{UMUS}_C$ of all MUSes of C and invocation of the underlying subroutines with G (i.e., $\mathtt{UMUS}_C$) instead of C. The usage of countInCell can be viewed as domain-specific instantiation of BSAT in the context of MUSes. Furthermore, we use the computed intersection of MUSes to improve the runtime efficiency of countInCell. It is perhaps worth mentioning that prior studies have observed that over 99% of the runtime of ApproxMC4 is spent inside the subroutine BSAT [Soos and Meel, 2019]. Therefore, the runtime efficiency of countInCell is crucial for the runtime performance of AMUSIC, and we discuss in detail, in Section 7.3.3, algorithmic contributions in the context of countInCell including usage of $\mathtt{IMUS}_C$.

We now argue that the replacement of C with G in Algorithm 7.1, line 4, does not affect correctness guarantees, which is stated formally below:

Proposition 7.2. *For every G' such that $\mathtt{UMUS}_C \subseteq G' \subseteq C$, the following hold:*

$$\mathtt{AMUS}_C = \mathtt{AMUS}_{G'} \tag{7.1}$$

$$\mathtt{IMUS}_C = \mathtt{IMUS}_{G'} \tag{7.2}$$

Proof. (1) Since $G' \subseteq C$ then every MUS of G' is also an MUS of C. In the other direction, every MUS of C is contained in the union $\mathtt{UMUS}_C$ of all MUSes of C, and thus every MUS of C is also an MUS of G' (since $G' \supseteq \mathtt{UMUS}_C$).
(2) $\mathtt{IMUS}_C = \bigcap_{M \in \mathtt{AMUS}_C} = \bigcap_{M \in \mathtt{AMUS}_{G'}} = \mathtt{IMUS}_{G'}$. □

```
1 $\mathcal{M} \leftarrow \{\}$
2 while $|\mathcal{M}| <$ threshold do
3     $M \leftarrow$ getMUS$(G, \mathtt{I}_G, \mathcal{M}, h, \alpha)$          // via a 3QBF solver, see Section 7.3.3
4     if $M =$ null then return $|\mathcal{M}|$
5     $\mathcal{M} \leftarrow \mathcal{M} \cup \{M\}$
6 return $|\mathcal{M}|$
```

Algorithm 7.4: countInCell$(G, \mathtt{I}_G, h, \alpha, \text{threshold})$

Equipped with Proposition 7.2, we now argue that each run of AMUSIC can be simulated by a run of ApproxMC4 for an appropriately chosen formula. In particular, given an unsatisfiable formula $C = \{f_1, \dots, f_{|C|}\}$, let us by B_C denote a satisfiable formula such that: (1) $Vars(B_C) = \{x_1, \dots, x_{|C|}\}$ and (2) an assignment $I : Vars(B_C) \rightarrow \{1, 0\}$ is a model of B_C iff $\{f_i | I(x_i) = 1\}$ is an MUS of C. Informally, models of B_C one-to-one map to MUSes of C. Hence, the size of sets returned by countInCell for C is identical to the corresponding BSAT for B_C. Since the analysis of ApproxMC4 only depends on the correctness of the size of the set returned by BSAT, we conclude that the answer computed by AMUSIC would satisfy the (ϵ, δ) guarantees. Furthermore, observing that AMUSIC performs $\lceil 17 \log_2(3/\delta) \rceil$ many iterations, each iterations performs $\mathcal{O}(\log |C|)$ many invocations of countInCell, and countInCell makes up to threshold $= 1 + 9.84(1 + \frac{\epsilon}{1+\epsilon})(1 + \frac{1}{\epsilon})^2$ many queries to Σ_3^P-oracle, we can bound the time complexity. Formally,

Proposition 7.3. *Given a formula C, a tolerance $\epsilon > 0$, and a confidence $1 - \delta \in (0, 1]$, let* AMUSIC(C, ϵ, δ) *return c. Then $Pr[|\mathtt{AMUS}_C|/(1 + \epsilon) \leqslant c \leqslant |\mathtt{AMUS}_C| \cdot (1 + \epsilon)] \geqslant 1 - \delta$. Furthermore,* AMUSIC *makes $\mathcal{O}(\log |C| \cdot \frac{1}{\epsilon^2} \cdot \log(1/\delta))$ calls to Σ_3^P oracle.*

Few words are in order concerning the complexity of AMUSIC. As noted at the beginning of this chapter (Chapter 7), given a formula with n variables, ApproxMC4 make $\mathcal{O}(\log n \cdot \frac{1}{\epsilon^2} \cdot \log(1/\delta))$ calls to an NP-oracle, whereas the task of finding a model of a formula requires a single call of an NP-oracle (a SAT solver). On the other hand, AMUSIC makes calls to a Σ_3^P-oracle (Section 7.3.3) while the problem of finding an MUS is in FP^{NP}. Therefore, a natural direction of future work is to design a hashing-based MUS counting technique that relies on an FP^{NP}-oracle.

7.3.3 *Counting MUSes in a Cell:* countInCell

In this section, we describe the procedure countInCell. The input of the procedure is the formula G (i.e., $\mathtt{UMUS}_C$), the set $\mathtt{I}_G = \mathtt{IMU}_G$, a hash function $h \in H_{xor}(|G|, m)$, a cell $\alpha \in \{0, 1\}^m$, and the threshold value. The output is the number $min(\text{threshold}, |\mathtt{AMU}_{\langle G,h,\alpha \rangle}|)$.

The description is provided in Algorithm 7.4. The algorithm iteratively searches for MUSes in the cell α and stores them to a set $\mathcal{M}$. To find each single MUS, the algorithm calls a procedure getMUS that returns either an MUS M such that $M \in (\texttt{AMU}_{\langle G,h,\alpha\rangle} \setminus \mathcal{M})$ or `null` if there is no such MUS. The algorithm terminates either when getMUS returns `null`, i.e. $\mathcal{M}$ contains all MUSes from α, or when $|\mathcal{M}| = \mathsf{threshold}$. The return value is the size of $\mathcal{M}$.

getMUS To implement the procedure getMUS, we build an $\exists\forall\exists$-QBF formula `MUSInCell` such that each witness of the formula corresponds to an MUS from $\texttt{AMU}_{\langle G,h,\alpha\rangle} \setminus \mathcal{M}$. The formula consists of several parts and uses several sets of variables that are described in the following.

The main part of the formula, shown in Equation (7.3), introduces the first existential quantifier and a set $P = \{p_1, \dots, p_{|G|}\}$ of variables that are quantified by the quantifier. Note that each valuation I of P corresponds to a subset S of G; in particular let us by $I_{P,G}$ denote the set $\{f_i \in G \,|\, I(p_i) = 1\}$. The formula is build in such a way that a valuation I is a witness of the formula if and only if $I_{P,G}$ is an MUS from $\texttt{AMU}_{\langle G,h,\alpha\rangle} \setminus \mathcal{M}$. This property is expressed via three conjuncts, denoted $\texttt{inCell}(P)$, $\texttt{unexplored}(P)$, and $\texttt{isMUS}(P)$, encoding that (i) $I_{P,G}$ is in the cell α, (ii) $I_{P,G}$ is not in $\mathcal{M}$, and (iii) $I_{P,G}$ is an MUS, respectively.

$$\texttt{MUSInCell} = \exists P.\, \texttt{inCell}(P) \wedge \texttt{unexplored}(P) \wedge \texttt{isMUS}(P) \tag{7.3}$$

Recall that the family $H_{xor}(n,m)$ of hash functions is defined as $\{h \,|\, h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \wedge y[k]) \mathit{for\,all}\, 1 \leqslant i \leqslant m\}$, where $a_{i,k} \in \{0,1\}$ (Section 7.1). A hash function $h \in H_{xor}(n,m)$ is given by fixing the values of individual $a_{i,k}$ and a cell α of h is a bit-vector from $\{0,1\}^m$. The formula $\texttt{inCell}(P)$ encoding that the set $I_{P,G}$ is in the cell α of h is shown in Equation (7.4).

$$\texttt{inCell}(P) = \bigwedge_{i=1}^{m} (a_{i,0} \oplus (\bigoplus_{p \in \{p_k | a_{i,k}=1\}} p) \oplus \neg\alpha[i]) \tag{7.4}$$

To encode that we are not interested in MUSes from $\mathcal{M}$, we can simply block all the valuations of P that correspond to these MUSes. However, we can do better. In particular, recall that if M is an MUS, then no proper subset and no proper superset of M can be an MUS; thus, we prune away all these sets from the search space. The corresponding formula is shown in Equation (7.5).

$$\texttt{unexplored}(P) = \bigwedge_{M \in \mathcal{M}} ((\bigvee_{f_i \in M} \neg p_i) \wedge (\bigvee_{f_i \notin M} p_i)) \tag{7.5}$$

The formula $\texttt{isMUS}(P)$ encoding that $I_{P,G}$ is an MUS is shown in Equation (7.6). Recall that $I_{P,G}$ is an MUS if and only if $I_{P,G}$ is unsatisfiable and for every *closest subset* S of $I_{P,G}$ it holds that S is satisfiable, where *closest subset* means that $|I_{P,G} \setminus S| = 1$. We encode these two conditions using two subformulas denoted by $\texttt{unsat}(P)$ and $\texttt{noUnsatSubset}(P)$.

$$\texttt{isMUS}(P) = \texttt{unsat}(P) \wedge \texttt{noUnsatSubset}(P) \tag{7.6}$$

The formula $\texttt{unsat}(P)$, shown in Equation (7.7), introduces the set $\mathit{Vars}(G)$ of variables that appear in G and states that every valuation of $\mathit{Vars}(G)$ falsifies at least one clause contained in $I_{P,G}$.

$$\texttt{unsat}(P) = \forall \mathit{Vars}(G).\ \bigvee_{f_i \in G} (p_i \land \neg f_i) \tag{7.7}$$

The formula $\texttt{noUnsatSubset}(P)$, shown in Equation (7.8), introduces another set of variables: $Q = \{q_1, \ldots, q_{|G|}\}$. Similarly as in the case of P, each valuation I of Q corresponds to a subset of G defined as $I_{Q,G} = \{f_i \in G \,|\, I(q_i) = 1\}$. The formula expresses that for every valuation I of Q it holds that $I_{Q,G}$ is satisfiable or $I_{Q,G}$ is not a closest subset of $I_{P,G}$.

$$\texttt{noUnsatSubset}(P) = \forall Q.\ \texttt{sat}(Q) \lor \neg\texttt{subset}(Q, P) \tag{7.8}$$

Equation (7.9) encodes the requirement that $I_{Q,G}$ is satisfiable. Since we are already reasoning about the satisfiability of G's clauses in Equation (7.7), we introduce here a copy G' of G where each variable x_i of G is substituted by its primed copy x_i'. Equation (7.9) states that there exists a valuation of $\mathit{Vars}(G')$ that satisfies $I_{Q,G}$.

$$\texttt{sat}(Q) = \exists \mathit{Vars}(G').\ \bigwedge_{f_i \in G'} (\neg q_i \lor f_i) \tag{7.9}$$

Equation (7.10) encodes that $I_{Q,G}$ is a closest subset of $I_{P,G}$. To ensure that $I_{Q,G}$ is a *subset* of $I_{P,G}$, we add the clauses $q_i \rightarrow p_i$. To ensure the *closeness*, we use cardinality constraints. In particular, we introduce another set $R = \{r_1, \ldots, r_{|G|}\}$ of variables and enforce their values via $r_i \leftrightarrow (p_i \land \neg q_i)$. Intuitively, the number of variables from R that are set to 1 equals to $|I_{P,G} \backslash I_{Q,G}|$. Finally, we add cardinality constraints, denoted by $\texttt{exactlyOne}(R)$, ensuring that exactly one r_i is set to 1.

$$\begin{aligned}\texttt{subset}(Q, P) = \exists R.\ &\bigwedge_{p_i \in P} ((q_i \rightarrow p_i) \land (r_i \leftrightarrow (p_i \land \neg q_i)) \\ &\land \texttt{exactlyOne}(R)\end{aligned} \tag{7.10}$$

Note that instead of encoding a *closest subset* in Equation 7.10, we could just encode that $I_{Q,G}$ is an arbitrary proper subset of $I_{P,G}$ as it would still preserve the meaning of Equation 7.6 that $I_{P,G}$ is an MUS. Such an encoding would not require introducing the set R of variables and also, at the first glance, would save a use of one existential quantifier. The thing is that the whole formula would still be in the form of $\exists\forall\exists$-QBF due to Equation 7.9 (which introduces the second existential quantifier). The advantage of using a closet subset is that we significantly prune the search space of the QBF solver. It is thus matter of contemporary QBF solvers whether it is more beneficial to reduce the number of variables (by removing R) or to prune the searchspace via R.

For the sake of lucidity, we have not exploited the knowledge of $\texttt{IMU}_G$ (I_G) while presenting the above equations. Since we know that every clause $f \in \texttt{IMU}_G$ has to be contained in every MUS of G, we can fix the values of the variables $\{p_i \,|\, f_i \in \texttt{IMU}_G\}$ to 1. This, in turn, significantly simplifies the equations and prunes away exponentially many (w.r.t. $|\texttt{IMU}_G|$) valuations of P, Q, and R, that need to be assumed. To solve the final formula, we employ a $\exists\forall\exists$-QBF solver, i.e., a Σ_3^P oracle.

One may wonder why we use our custom solution for identifying MUSes in a cell instead of employing one of existing MUS extraction techniques. Conventional MUS extraction algorithms cannot be used to identify MUSes that are in a cell since the cell is not "continuous" w.r.t. the set containment. In particular, assume that we have three sets of clauses, K, L, M, such that $K \subsetneq L \subsetneq M$. It can be the case that K and M are in the cell, but L is not in the cell. Contemporary MUS extraction techniques require the search space to be continuous w.r.t. the set containment and thus cannot be used in our case.

Finally, observing that AMUSIC performs multiple iterations, and especially multiple calls of the procedure countInCell, one might think of storing all the MUSes AMUSIC already identified into a set *knownMUSes*, and then use *knownMUSes* to speed up the further computation. In particular, whenever we call countInCell to count MUSes in a cell α, we could first check if some MUSes of *knownMUSes* are in the cell and hence possibly save some invocations of the QBF solver. We actually tried this and it brought no improvement; the thing is that we assume exponentially many cells w.r.t. $|C|$ and hence the probability that an MUS repeatedly falls into a cell we examine is extremely low. Another possible use of *knownMUSes* would be to prune the search space of the QBF solver when solving the QBF encoding. In particular, like in Equation (7.5), we could remove all subsets and all supersets of the already known MUSes from the search space. We have also tried this, however, it actually slowed down the computation.

7.3.4 *Computing* $\texttt{UMUS}_C$

We now turn our attention to computing the union $\texttt{UMUS}_C$ (i.e., G) of all MUSes of C. Let us start by describing well-known concepts of *autark variables* and a *lean kernel*. A set $A \subseteq \mathit{Vars}(C)$ of variables is an *autark* of C iff there exists a truth assignment to A such that every clause of C that contains a variable from A is satisfied by the assignment [Monien and Speckenmeyer, 1985]. It holds that the union of two autark sets is also an autark set, thus there exists a unique largest autark set (see, e.g., [Kleine Büning and Kullmann, 2009, Kullmann, 2000b]). The *lean kernel* of C is the set of all clauses that do not contain any variable from the largest autark set. It is known that the *lean kernel* of C is an over-approximation of $\texttt{UMUS}_C$ (see e.g., [Kleine Büning and Kullmann, 2009, Kullmann, 2000b]), and there were proposed several algorithms, e.g., [Marques-Silva et al., 2014, Kullmann and Marques-Silva, 2015]), for computing the lean kernel.

```
1 $K \leftarrow$ the lean kernel of $C$; $\mathcal{M} \leftarrow \{\}$
2 for $f \in K \setminus \{f \in M \,|\, M \in \mathcal{M}\}$ do
3     $W \leftarrow \mathsf{checkNecessity}(f, K)$                // via the QBF encoding in Eq. 7.11
4     if $W \neq$ null then $\mathcal{M} \leftarrow \mathcal{M} \cup \{$an MUS of $W\}$
5     else $K \leftarrow K \setminus \{f\}$
6 return $K$
```

Algorithm 7.5: $\mathsf{getUMU}(C)$

Algorithm Our approach for computing $\mathtt{UMUS}_C$ consists of two steps. First, we compute the lean kernel K of C to get an over-approximation of $\mathtt{UMUS}_C$, and then we gradually refine the over-approximation K until K is exactly the set $\mathtt{UMUS}_C$. The refinement is done by solving the MUS-membership problem for each $f \in K$. To solve the MUS-membership problem efficiently, we reveal a connection to critical clauses, as stated in the following proposition.

Proposition 7.4. *A clause $f \in C$ belongs to $\mathtt{UMUS}_C$ iff there is a subset W of C such that W is unsatisfiable and f is critical for W (i.e., $W \setminus \{f\}$ is satisfiable).*

Proof. $\Rightarrow$: Let $f \in \mathtt{UMUS}_C$ and $M \in \mathtt{AMUS}_C$ such that $f \in M$. Since M is an MUS then $M \setminus \{f\}$ is satisfiable; thus f is critical for M.
$\Leftarrow$: If W is a subset of C and $f \in W$ a critical clause for W then f has to be contained in every MUS of W. Moreover, W has at least one MUS and since $W \subseteq C$, then every MUS of W is also an MUS of C. □

Our approach for computing $\mathtt{UMUS}_C$ is shown in Algorithm 7.5. It takes as an input the formula C and outputs $\mathtt{UMUS}_C$ (denoted K). Moreover, the algorithm maintains a set $\mathcal{M}$ of MUSes of C. Initially, $\mathcal{M} = \emptyset$ and K is set to the lean kernel of C; we use an approach by Marques-Silva et al. [Marques-Silva et al., 2014] to compute the lean kernel. At this point, we know that $K \supseteq \mathtt{UMUS}_C \supseteq \{f \in M \,|\, M \in \mathcal{M}\}$. To find $\mathtt{UMUS}_C$, the algorithm iteratively determines for each $f \in K \setminus \{f \in M \,|\, M \in \mathcal{M}\}$ if $f \in \mathtt{UMUS}_C$. In particular, for each f, the algorithm checks whether there exists a subset W of K such that f is critical for W (Proposition 7.4). The task of finding W is carried out by a procedure $\mathsf{checkNecessity}(f, K)$. If there is no such W, then the algorithm removes f from K. In the other case, if W exists, the algorithm finds an MUS of W and adds the MUS to the set $\mathcal{M}$. Any available single MUS extraction approach, e.g., [Belov and Marques-Silva, 2012, Bacchus and Katsirelos, 2015, Belov et al., 2014, Nadel et al., 2014], can be used to find the MUS.

To implement the procedure $\mathsf{checkNecessity}(f, K)$ we build a QBF formula that is true iff there exists a set $W \subseteq K$ such that W is unsatisfiable and f is critical for W. To represent W we introduce a set $S = \{s_g \,|\, g \in K\}$ of Boolean variables; each valuation I of S corresponds to a subset $I_{S,K}$ of K defined as $I_{S,K} = \{g \in K \mid I(s_g) = 1\}$. Our encoding is shown in Equation 7.11.

$$\exists S, \mathit{Vars}(K). \forall \mathit{Vars}(K'). s_f \wedge (\bigwedge_{g \in K \setminus \{f\}} (g \vee \neg s_g)) \wedge (\bigvee_{g \in K'} (\neg g \wedge s_g)) \tag{7.11}$$

The formula consists of three main conjuncts. The first conjunct ensures that f is present in $I_{S,K}$. The second conjunct states that $I_{S,K} \setminus \{f\}$ is satisfiable, i.e., that there exists a valuation of $\mathit{Vars}(K)$ that satisfies $I_{S,K} \setminus \{f\}$. Finally, the last conjunct expresses that $I_{S,K}$ is unsatisfiable, i.e., that every valuation of $\mathit{Vars}(K)$ falsifies a clause of $I_{S,K}$. Since we are already reasoning about variables of K in the second conjunct, in the third conjunct, we use a primed version (a copy) K' of K.

Alternative QBF Encodings Janota and Marques-Silva [Janota and Marques-Silva, 2011] proposed three other QBF encodings for the MUS-membership problem, i.e., for deciding whether a given clause f of a formula K belongs to an MUS of K. Two of the three proposed encodings are typically inefficient, thus we describe the two only briefly. The third encoding is described in more detail.

The first encoding by Janota and Marques-Silva uses 3 levels of quantifiers and it directly express that there exists an MUS S of K that contains f. Intuitively: "there **exists** a subset S of K such that $f \in S$ and **all** valuations of $\mathit{Vars}(K)$ falsify S (i.e., S is unsat) and **for all** proper subsets of S there **exist** valuations that satisfy the subsets".

Their second encoding uses only 2 levels of quantifiers, but it requires a quadratic number of variables w.r.t. $|K|$. The encoding is very similar to the first encoding, however, instead of expressing that all proper subsets of S are satisfiable, it express that all *closest* proper subsets of S are satisfiable (i.e., the subsets that differ only in one clause). There are only linearly many closest subsets of S, thus they can be enumerated in the formula.

Finally, their last and the best encoding uses only 2 levels of quantifiers and only a linear number of variables w.r.t. $|K|$; we describe this one in more detail. This encoding is based on the minimal hitting set duality between MUSes and MCSes (Proposition 2.6). Due to the duality, it holds that a clause $f \in K$ belongs to an MUS of K if and only if f belongs to an MCS of K. Recall that every MCS of K is a complement of an MSS of K. The encoding by Janota and Marques-Silva expresses that there exists a subset D of K such that $f \notin D$, D is satisfiable, and for all D' such that $D \subsetneq D' \subseteq K$, the set D' is unsatisfiable (i.e., D is an MSS). To encode the sets D and D', they introduce two additional sets of variables: $R = \{r_g \mid g \in K\}$ and $R' = \{r'_g \mid g \in K\}$. In particular, each valuation I of R corresponds to the set $I^-_{R,K} = \{g \in K \mid I(r_g) = 0\}$, and similarly each valuation I of R' corresponds to the set $I^-_{R',K} = \{g \in K \mid I(r'_g) = 0\}$[2]. The QBF encoding

[2] Note that in our encoding, in Equation 7.11, we use in a similar manner the variables S to introduce a set $I_{S,K}$. However, we use S as *activation* variables (i.e., $g \in I_{S,K}$ iff $I(s_g) = 1$), whereas here the variables R are used as *relaxation* variables (i.e., $g \in I^-_{R,K}$ iff $I(r_g) = 0$). The purpose (and the effect) of the activation and relaxation variables is the same: they encode a subset of K.

is shown in Equations (7.12) and (7.13).

$$\begin{aligned}\exists R, \mathit{Vars}(K). \forall R', \mathit{Vars}(K'). \, r_f \wedge (\bigwedge_{g \in K} (g \vee r_g)) \\ \wedge (R' < R \rightarrow (\bigvee_{g \in K'} (\neg g \wedge \neg r'_g)))\end{aligned} \tag{7.12}$$

$$R' < R \equiv \bigwedge_{r_i \in R} r_i \rightarrow r'_i \wedge \bigvee_{r_i \in R} \neg r_i \wedge r'_i \tag{7.13}$$

Equation (7.12) consists of three main conjuncts. The first conjunct states that $r_f \notin I^-_{R,K}$ and the second conjunct states that $I^-_{R,K}$ is satisfiable. The third conjunct expresses that if $I^-_{R,K} \subsetneq I^-_{R',K'}$, then $I^-_{R',K}$ is unsatisfiable. To avoid a clash with the second conjunct, a primed version K' of K (i.e., a copy) is used in the third conjunct. The property that $I^-_{R,K} \subsetneq I^-_{R',K}$ is described via the formula $R' < R$ (Equation 7.13).

Compared to our encoding (Equation 7.11), both the encodings use the same quantifiers; however, our encoding is smaller. In particular, the encoding by Janota and Marques-Silva uses $2 \times (\mathit{Vars}(K) + |K|)$ variables whereas our encoding uses only $|K| + 2 \times \mathit{Vars}(K)$ variables, and it leads to smaller formulas.

Implementation Recall that we compute $\texttt{UMUS}_C$ to reduce the search space, i.e. instead of working with the whole C, we work only with $G = \texttt{UMUS}_C$. The soundness of this reduction is witnessed in Proposition 7.2 (Section 7.3.2). In fact, Proposition 7.2 shows that it is sound to reduce the search space to any G' such that $\texttt{UMUS}_C \subseteq G' \subseteq C$. Since our algorithm for computing $\texttt{UMUS}_C$ subsumes repeatedly solving a Σ_2^P-complete problem, it can be very time-consuming. Therefore, instead of computing the exact $\texttt{UMUS}_C$, we optionally compute only an over-approximation G' of $\texttt{UMUS}_C$. In particular, we allow the user of our algorithm to set a time limit for computing the lean kernel K of C. Moreover, we use a time limit for executing the procedure $\mathsf{checkNecessity}(f, K)$; if the time limit is exceeded for a clause $f \in K$, we conservatively assume that $f \in \texttt{UMUS}_C$, i.e., we over-approximate.

7.3.5 *Computing* $\texttt{IMU}_G$

Our approach to compute the intersection $\texttt{IMU}_G$ (i.e., I_G) of all MUSes of G is composed of several ingredients. First, recall that a clause $f \in G$ belongs to $\texttt{IMU}_G$ iff f is critical for G. Another ingredient is the ability of contemporary SAT solvers to provide either a model or an *unsat core* of a given unsatisfiable formula $N \subseteq G$, i.e., a small, yet not necessarily minimal, unsatisfiable subset of N. The final ingredient is a technique called *model rotation*. The technique was originally proposed by Marques-Silva and Lynce [Marques-Silva and Lynce, 2011], and it serves to explore critical clauses based on other already known critical clauses. In particular, let f be a critical clause for G and $I : \mathit{Vars}(G) \rightarrow \{0,1\}$ a model of $G \setminus \{f\}$. Since G is unsatisfiable, the model I does not satisfy f. The model rotation attempts to alter

```
cands ← $G$
$K \leftarrow \emptyset$
while $cands \neq \emptyset$ do
    $f \leftarrow$ choose $f \in cands$
    $(sat?, I, core) \leftarrow \mathsf{isSAT}(G \setminus \{f\})$    // $I$ is a model or $core$ is an unsat core of $(G \setminus \{f\})$
    if $sat?$ then
        $R \leftarrow \mathsf{RMR}(G, f, I)$    // external recursive model rotation procedure
        $K \leftarrow K \cup \{f\} \cup R$
        $cands \leftarrow cands \setminus (\{f\} \cup R)$
    else
        $cands \leftarrow cands \cap core$
return $K$
```

Algorithm 7.6: $\mathsf{getIMU}(G)$

I by switching, one by one, the Boolean assignment to the variables $Vars(\{f\})$. Each variable assignment I' that originates from such an alternation of I necessarily satisfies f and does not satisfy at least one $f' \in G$. If it is the case that there is exactly one such f', then f' is critical for G. An improved version of model rotation, called *recursive model rotation*, was later proposed by Belov and Marques-Silva [Belov and Marques-Silva, 2011] who noted that the model rotation could be recursively performed on the newly identified critical clauses.

Our approach for computing $\mathtt{IMU}_G$ is shown in Algorithm 7.6. To find $\mathtt{IMU}_G$, the algorithm determines for each f whether f is critical for G. In particular, the algorithm maintains two sets: a set *cands* of *candidates* on critical clauses and a set K of already known critical clauses. Initially, K is empty and $cands = G$. At the end of computation, *cands* is empty and K equals to $\mathtt{IMU}_G$. The algorithm works iteratively. In each iteration, the algorithm picks a clause $f \in cands$ and checks $G \setminus \{f\}$ for satisfiability via a procedure isSAT. Moreover, isSAT returns either a model I or an unsat core *core* of $G \setminus \{f\}$. If $G \setminus \{f\}$ is satisfiable, i.e. f is critical for G, the algorithm employs the recursive model rotation, denoted by $\mathsf{RMR}(G, f, I)$, to identify a set R of additional critical clauses. Subsequently, all the newly identified critical clauses are added to K and removed from *cands*. In the other case, when $G \setminus \{f\}$ is unsatisfiable, the set *cands* is reduced to $cands \cap core$ since every critical clause of G has to be contained in every unsatisfiable subset of G. Note that $f \notin core$, thus at least one clause is removed from *cands*.

7.4 *Experimental Evaluation*

We employed several external tools to implement AMUSIC. In particular, we use the QBF solver CAQE [Rabe and Tentrup, 2015] for solving the QBF formula `MUSInCell`, the 2QBF solver CADET [Rabe et al., 2018] for solving our ∃∀-QBF encoding while computing $\mathtt{UMUS}_C$, and the QBF preprocessor QRATPre+ [Lonsing and Egly, 2019] for preprocessing/simplifying our QBF encodings. Moreover, we employ muser2 [Belov and Marques-Silva, 2012] for a single MUS extraction while computing $\mathtt{UMUS}_C$, a MaxSAT solver UWrMaxSat [Piotrów, 2019] to implement the algorithm by Marques-Silva et al. [Marques-Silva et al., 2014] for computing the lean kernel of C, and finally, we use a toolkit called PySAT [Ignatiev et al., 2018] for encoding cardinality constraints used in the formula `MUSInCell`. The tool along with all benchmarks that we used is available at:

https://github.com/jar-ben/amusic

Objectives As noted earlier, AMUSIC is the first technique to (approximately) count MUSes without explicit enumeration. We demonstrate the efficacy of our approach via a comparison with two state of the art techniques for MUS enumeration: MARCO [Liffiton et al., 2016] and MCSMUS [Bacchus and Katsirelos, 2016]. Within a given time limit, an MUS enumeration algorithm either identifies the whole $\mathtt{AMUS}_C$, i.e., provides the exact value of $|\mathtt{AMUS}_C|$, or identifies just a subset of $\mathtt{AMUS}_C$, i.e., provides an under-approximation of $|\mathtt{AMUS}_C|$ with no approximation guarantees.

Our experimental evaluation has two parts. First, we examine the *empirical accuracy* of AMUSIC, i.e., the exactness of the provided MUS counts. Second, we experimentally examine the scalability of AMUSIC, MARCO, and MCSMUS w.r.t. $|\mathtt{AMUS}_C|$.

Benchmarks And Experimental Setup To be able to answer our research questions, we have somewhat contradicting requirements on the experimental benchmarks. On one hand, to evaluate the (ϵ, δ) guarantees of our algorithm, we need benchmarks for which the number of contained MUSes is known. On the other hand, to find out what are the limits of MUS enumeration algorithms, we need benchmarks where a complete MUS enumeration is practically intractable.

Unfortunately, we are not aware of any publicly available benchmarks that would fulfill our requirements. A vast majority of papers on MUS enumeration/extraction use the benchmarks from the MUS track of the SAT 2011 Competition that we used in Chapters 4, 5 and 6. Although most of these benchmarks are intractable for a complete MUS enumeration, the number of MUSes in these benchmarks is unknown. Moreover, these benchmarks contain thousands or even millions of clauses, and since our approach relies on a Σ_3^P oracle (3QBF solver), AMUSIC is not able to handle inputs with so many clauses.

Therefore, we generated a custom collection of scalable benchmarks. The benchmarks mimic requirements on multiprocessing systems. Assume that we are given a system with two groups (kinds) of processes, $A = \{a_1, \ldots, a_{|A|}\}$ and $B = \{b_1, \ldots, b_{|B|}\}$, such that $|A| \geqslant |B|$. The processes require resources of the system; however, the resources are limited. Therefore, there are restrictions on which processes can be active simultaneously. In particular, we have the following three types of mutually independent restrictions on the system:

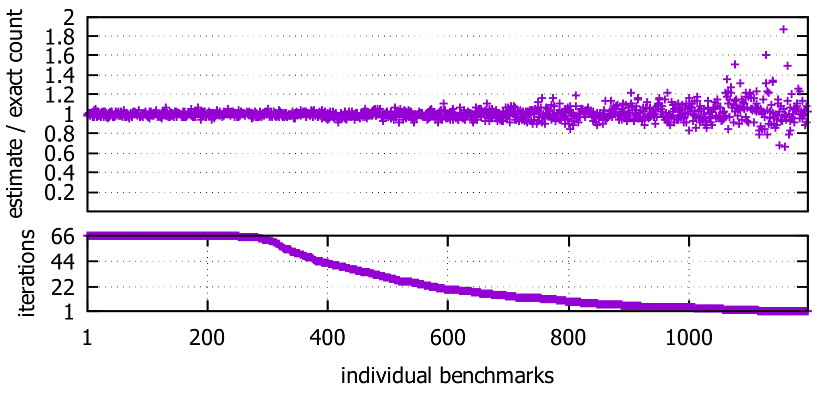


Figure 7.2: The number of completed iterations and the accuracy of the final MUS count estimate for individual benchmarks.

- The first type of restriction states that "at most $k-1$ processes from the group A can be active simultaneously", where $k \leqslant |A|$.
- The second type of restriction enforces that "if no process from B is active then at most $k-1$ processes from A can be active, and if at least one process from B is active then at most $l-1$ processes from A can be active", where $k, l \leqslant |A|$.
- The third type of restriction includes the second restriction. Moreover, we assume that a process from B can activate a process from A. In particular, for every $b_i \in B$, we assume that when b_i is active, then a_i is also active.

We encode the three restrictions via three Boolean CNF formulas, R_1, R_2, R_3. The formulas use three sets of variables: $X = \{x_1, \ldots, x_{|A|}\}$, $Y = \{y_1, \ldots, y_{|B|}\}$, and Z. The sets X and Y represent the Boolean information about activity of processes from A and B: a_i is active iff $x_i = 1$ and b_j is active iff $y_j = 1$. The set Z contains additional auxiliary variables. Moreover, we introduce a formula $\mathtt{ACT} = (\bigwedge_{x_i \in X} x_i) \wedge (\bigwedge_{y_i \in Y} y_i)$ encoding that all processes are active. For each $i \in \{1, 2, 3\}$, the conjunction $G_i = R_i \wedge \mathtt{ACT}$ is unsatisfiable. Intuitively, every MUS of G_i represents a minimal subset of processes that need to be active to violate the restriction. The number of MUSes in G_1, G_2, and G_3 is $\binom{|A|}{k}$, $\binom{|A|}{k} + |B| \times \binom{|A|}{l}$, and $\binom{|A|}{k} + \sum_{i=1}^{|B|} (\binom{|B|}{i} \times \binom{|A|-1}{l-i})$, respectively. We generated G_1, G_2, and G_3 for these values: $10 \leqslant |A| \leqslant 30$, $2 \leqslant |B| \leqslant 6$, $\left\lfloor \frac{|A|}{2} \right\rfloor \leqslant k \leqslant \left\lfloor \frac{3 \times |A|}{2} \right\rfloor$, and $l = k - 1$. In total, we obtained 1353 benchmarks (formulas) that range in their size from 78 to 361 clauses, use from 40 to 152 variables, and contain from 120 to 1.7×10^9 MUSes.

All experiments were run using a time limit of 7200 seconds and computed on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux 4.19.67-2. The values of ϵ and δ were set to 0.8 and 0.2, respectively. Complete results are available in the online appendix[3].

[3] https://www.fi.muni.cz/~xbendik/phdThesis/

Accuracy Recall that to compute an estimate c of $|\mathtt{AMUS_C}|$, AMUSIC performs multiple iteration of executing AMUSICCore to get a list S of multiple estimates of $|\mathtt{AMUS_C}|$, and then use the median of S as the

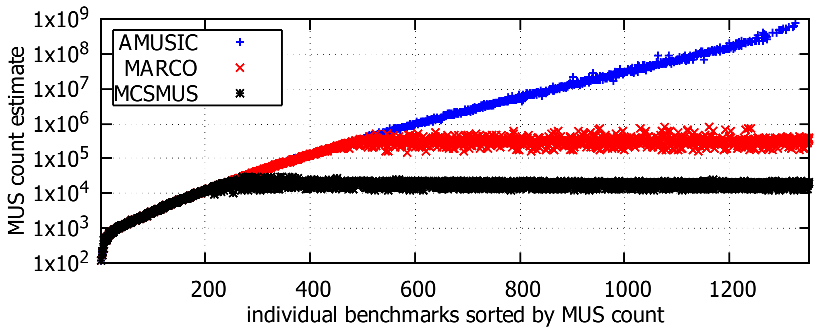


Figure 7.3: Scalability of AMUSIC, MARCO, and MCSMUS w.r.t. $|\mathtt{AMUS}_C|$.

final estimate c. The more iterations are performed, the higher is the confidence that c is within the required tolerance $\epsilon = 0.8$, i.e., that $\frac{|\mathtt{AMUS}_C|}{1.8} \leqslant c \leqslant 1.8 \cdot |\mathtt{AMUS}_C|$. To achieve the confidence $1 - \delta = 0.8$, 66 iterations need to be performed. In case of 157 benchmarks, the algorithm was not able to finish even a single iteration, and only in case of 251 benchmarks, the algorithm finished all the 66 iterations. For the remaining 945 benchmarks, at least some iterations were finished, and thus at least an estimate with a lower confidence was determined.

We illustrate the achieved results in Figure 7.2. The figure consists of two plots. The plot at the bottom of the figure shows the number of finished iterations (y-axis) for individual benchmarks (x-axis). The plot at the top of the figure shows how accurate were the MUS count estimates. In particular, for each benchmark (formula) C, we show the number $\frac{c}{|\mathtt{AMUS}_C|}$ where c is the final estimate (median of estimates from finished iterations). For benchmarks where all iterations were completed, it was always the case that the final estimate is within the required tolerance, although we had only 0.8 theoretical confidence that it would be the case. Moreover, the achieved estimate never exceeded a tolerance of 0.1, which is much better than the required tolerance of 0.8. As for the benchmarks where only some iterations were completed, there is only a single benchmark where the required tolerance of 0.8 was exceeded.

Scalability The scalability of AMUSIC, MARCO, and MCSMUS w.r.t. the number of MUSes ($|\mathtt{AMUS}_C|$) is illustrated in Figure 7.3. In particular, for each benchmark (x-axis), we show in the plot the estimate of the MUS count that was achieved by the algorithms (y-axis). The benchmarks are sorted by the exact count of MUSes in the benchmarks. MARCO and MCSMUS were able to finish the MUS enumeration, and thus to provide the count, only for benchmarks that contained at most 10^6 and 10^5 MUSes, respectively. AMUSIC, on the other hand, was able to provide estimates on the MUS count even for benchmarks that contained up to 10^9 MUSes. Moreover, as we have seen in Figure 7.2, the estimates are very accurate. Only in the case of 157 benchmarks where AMUSIC finished no iteration, it could not provide any estimate.

7.5 *Summary and Future Work*

We presented a probabilistic algorithm, called AMUSIC, for approximate MUS counting that needs to explicitly identify only logarithmically many MUSes and yet still provides strong theoretical guarantees. The high-level idea is adopted from a model counting algorithm ApproxMC4: we partition the search space into small cells, then count MUSes in a single cell, and estimate the total count by scaling the count from the cell. The novelty lies in the low-level algorithmic parts that are specific for MUSes. Mainly, (1) we propose QBF encoding for counting MUSes in a cell, (2) we exploit MUS intersection to speed-up localization of MUSes, and (3) we utilize MUS union to reduce the search space significantly. Our experimental evaluation showed that the scalability of AMUSIC w.r.t. the MUS count outperforms the scalability of contemporary enumeration-based counters by several orders of magnitude. Moreover, the practical accuracy of AMUSIC is significantly better than what is guaranteed by the theoretical guarantees. On the other hand, AMUSIC does not scale very well w.r.t. the number of clauses in the input formula, since the number of clauses affects the size of the $\exists\forall\exists$-QBF formula `MUSInCell`.

Our work opens up several questions at the intersection of theory and practice. From a theoretical perspective, the natural question is to ask if we can design a scalable approximate MUS counting algorithm that makes polynomially many calls to an *NP* oracle. From a practical perspective, our work showcases interesting applications of QBF solvers with native XOR support. Since approximate counting and sampling are known to be inter-reducible, another line of work would be to investigate the development of an almost-uniform sampler for MUSes. Another possible direction of future work is to extend our MUS counting approach to other instances of the general concept of minimal sets over a monotone predicate.

8
Boolean CNF MSS and MCS Counting

This is the last chapter of the thesis that focuses on analyzing a given unsatisfiable set C of Boolean clauses. In particular, the problem we deal with here is counting the number of maximal satisfiable subsets (MSSes) of C. Equivalently, since MSSes are the complements of the minimal correction subsets (MCSes) of C, we also deal with the problem of counting MCSes of C. On contrary to the previous chapter where we focused on *approximate counting* of minimal unsatisfiable subsets (MUSes), here we provide an approach [Bendík and Meel, 2021] for *exact counting* of MSSes (and MCSes).

The progress in the development of MSS/MCS identification techniques mirrors the progress in the development of techniques for identifying minimal unsatisfiable subsets (discussed in Chapter 5). Since scalable techniques for identification of MSSes appeared only about a decade and a half ago, the earliest applications primarily focused on a reduction to the identification of a single MSS or a small set of MSSes (or MCSes). With recent improvements in the scalability of MSS identification techniques, research now started with examining additional MSS related problems and their corresponding applications. One of such possible research directions is the problem of counting MSSes of a given unsatisfiable CNF formula.

Our interest in counting the number of MSSes is motivated by the rise of Beyond NP paradigm wherein the progress in the design of efficient techniques for satisfiability paved the way for interest in the design of efficient techniques for problems such as counting, sampling, optimization, and the like. In particular, the past two decades have witnessed a proliferation of efficient techniques for model counting, also denoted as #SAT. It is worth remarking that initial studies into model counting were motivated by applications in Bayesian inference but the subsequent availability of efficient model counting techniques have now led to several new applications ranging from neural network verification [Baluta et al., 2019], quantified information flow [Biondi et al., 2018], computational biology [Sashittal and El-Kebir, 2020], network reliability, and the like. In this regard, we view that given the availability of efficient techniques for finding an MSS/MCS, it is the right time to pursue an investigation into the design of efficient counting techniques for MSSes/MCSes, and the availability of efficient counting techniques for MCSes/MSSes would lead to a discovery of a diverse set of applications for MSS counting. So far, we are aware just of a single application of MSS counting which emerges in the field of diagnosis where the MSS count serves as a good diagnostic metric [Thimm, 2018].

Similarly to the early years of research into #SAT, the best-known technique, as of now, to perform the MSS counting is to employ state of the art techniques for complete MSS enumeration. While MSS enumeration techniques have improved over the years, the complete MSS enumeration is often practically intractable since there can be up to exponentially many MSSes w.r.t. the size of the input constraint set. In this context, the primary research question that we seek to investigate is *whether we can design MSS counting techniques that do not necessarily rely on enumeration?*. We envision the development of MSS counting techniques to take advantage of the progress in the model counting techniques. Given that the problem of finding an MSS is in $\mathrm{FP}^{\mathrm{NP}[\log n]}$ [Janota and Marques-Silva, 2016] (i.e., harder than the classical SAT problem), a natural target problem is projected model counting, a generalization of the classical model counting problem; the projected model counting is known to be #NP-hard in contrast to #P-completeness of classical model counting.

The primary contribution of this chapter is an affirmative answer to the above question. In particular, we design a new algorithmic framework that uses a novel architecture of a wrapper $\mathcal{W}$ and a remainder $\mathcal{R}$ such that the desired MSS count corresponding to the formula C is $|\mathcal{W}| - |\mathcal{R}|$. We encode the wrapper $\mathcal{W}$ and the remainder $\mathcal{R}$ via Boolean formulas $\mathbb{W}$ and $\mathbb{R}$ with suitable projection sets such that the projected model count of $\mathbb{W}$ and $\mathbb{R}$ is equal to $|\mathcal{W}|$ and $|\mathcal{R}|$ respectively. We present four different strategies for the construction of wrappers (and their corresponding remainders) and observe the soundness of a composition of different wrappers. The reduction to projected model counting allows us to build on recent advances in the design of efficient component caching-based projected model counting techniques. To demonstrate the empirical effectiveness of our approach, we implemented a Python-based prototype and performed a detailed empirical analysis. Out of 1200 benchmarks, the enumeration-based techniques can solve 353 benchmarks while our approach can solve 510 benchmarks.

The chapter is organized as follows. Section 8.1 defines the basic concepts and notation that is specific for this chapter. Subsequently, Section 8.2 provides an overview of related work. Our novel technique for counting MSSes is presented in Section 8.3. Finally, Section 8.4 discusses results of our experimental evaluation.

8.1 Prelimilaries and Problem Formulation

We assume standard definitions and notation for propositional logic as defined in Section 2. Furthermore, for the sake of this chapter, we define the following extensions. Given a set A of variables, a valuation π of A, and a formula F, we write $F[\pi]$ to denote the substitution of each variable x in the domain of π by the value $\pi(x)$; furthermore, we apply trivial simplifications, e.g., $G \vee \mathit{False} = G$, $G \wedge \mathit{False} = \mathit{False}$, etc. Observe that if $A \supseteq \mathit{Vars}(F)$, then $F[\pi]$ is simplified either to *True* or to *False*. We write M_F to denote the set of all models of a formula F. Furthermore, for a set A of variables such that $A \subseteq \mathit{Vars}(F)$, we write $M_{F\downarrow A}$ to denote the projection of M_F on A, and for $\pi \in M_F$, we write $\pi_{\downarrow A}$ to denote the projection of π on A. Two formulas F and G are *equivalent*, denoted $F \equiv G$, iff $M_F = M_G$. Finally, we write $\mathit{Vals}(F)$ to denote the set of all valuations of the variables $\mathit{Vars}(F)$ of F.

Given an unsatisfiable CNF formula C, we write $\texttt{MSS}_C$ to denote the set of all MSSes of C, $\texttt{MCS}_C$ to denote the set of all MCSes of C, $\texttt{SS}_C$ to denote the set of all satisfiable subsets of C, and $\texttt{nonMSS}_C$ to denote the set $\texttt{SS}_C \backslash \texttt{MSS}_C$ of all satisfiable subsets of C that are not MSSes of C.

In this chapter, we are concerned with the following three problems.

Name: #MSS
Input: A formula C.
Output: The number $|\texttt{MSS}_C|$ of MSSes of C.

Name: #MCS
Input: A formula C.
Output: The number $|\texttt{MCS}_C|$ of MCSes of C.

Name: proj-#SAT
Input: A formula C and a set of variables $S \subseteq \mathit{Vars}(C)$.
Output: The number $M_{C\downarrow S}$ of models of C projected on S.

Our goal is to solve the #MCS and #MSS problems. Since MCSes are complements of MSSes, the two problems are equivalent. Thus, in the following, we focus only on the #MSS problem. Finally, we do not focus on solving the proj-#SAT problem; instead, we propose several reductions of #MSS to proj-#SAT.

8.2 Related Work

MSS Counting As far as we know, there is no algorithm dedicated to counting MSSes. A straightforward approach to determine the count is to enumerate all the MSSes via an MSS enumeration algorithm, e.g., [Bailey and Stuckey, 2005, Stern et al., 2012, Liffiton et al., 2016, Marques-Silva et al., 2013a, Bendík et al., 2016b, Narodytska et al., 2018, Previti et al., 2018, Bendík and Černá, 2020b], and then simply count the enumerated MSSes. However, the complete MSS enumeration is often practically intractable, since there can be exponentially many MSSes w.r.t. $|C|$ and thus, the MSSes just cannot be explicitly enumerated in a reasonable time.

Another possible solution how to count MSSes (MCSes) is based on the minimal hitting set duality between minimal unsatisfiable subsets and MCSes of C (Observation 2.6). In particular, one can first use an MUS enumeration algorithm, e.g., [Bailey and Stuckey, 2005, Stern et al., 2012, Liffiton et al., 2016, Bacchus and Katsirelos, 2015, 2016, Narodytska et al., 2018, Bendík et al., 2018b, Liu and Luo, 2018], to identify all MUSes of C, and then count the number of minimal hitting sets of the MUSes. The problem is that there can be also exponentially many MUSes w.r.t. $|C|$, which makes the MUS enumeration also often practically intractable.

Due to the duality between MUSes and MCSes, a tightly connected problem is also the MUS counting which we have studied in Chapter 7. However, our MUS counting algorithm AMUSIC is *only* a probabilistic approximate counter. In contrast, here we focus on the exact MSS counting. Moreover, although the notions of MUSes and MSSes share a close relationship, we are unaware of any efficient reduction of the MSS counting problem to the MUS counting problem.

Model Counting In [Valiant, 1979], it was shown that the problem of propositional model counting (i.e., `proj-#SAT` when $S = Vars(C)$) is #P-complete. The `proj-#SAT` problem was shown [Durand et al., 2005] to be $\#NP$-hard. From a practical perspective, the earliest work on model counting dates to [Birnbaum and Lozinskii, 1999], which sought to rely on *smarter* enumeration strategies of partial solutions. Subsequently, [Bayardo Jr. and Pehoushek, 2000] introduced the notion of component caching, wherein a residual formula after substituting the current partial assignment can be partitioned into different subsets of clauses such that these subsets do not share variables. Each of these subsets is called a component, and the model count of the formula is obtained by multiplying the corresponding counts for each of the components. Therefore, the model count is often determined by explicitly identifying only a fraction of all models. Caching scheme is used to avoid recomputation as similar components appear in different parts of the search space. Over the past two decades, there has been a series of algorithmic and system-driven improvements of component caching-based model counting techniques [Sang et al., 2004, 2005, Thurley, 2006, Muise et al., 2012, Sharma et al., 2019]

Over the past 5 years, there has been a concentrated effort on developing efficient projected model counting techniques [Chakraborty et al., 2014, Aziz et al., 2015, Chakraborty et al., 2016, Möhle and Biere, 2018, Sharma et al., 2019, Lagniez and Marquis, 2019]. These techniques rely on appropriate modifications of standard propositional model counters (as described above). In this work, we rely on the state of the art projected model counter GANAK [Sharma et al., 2019], which was part of the system that won the Projected model counting track at the recently organized model counting competition[1].

[1] https://mccompetition.org/

8.3 Counting the Number of MSSes

8.3.1 Basic Idea

Our approach for finding the MSS count $|\texttt{MSS}_C|$ is based on a simple observation: one can count the number $|\texttt{SS}_C|$ of all satisfiable subsets of C, the number $|\texttt{nonMSS}_C|$ of satisfiable subsets that are not MSSes, and then do the math: $|\texttt{MSS}_C| = |\texttt{SS}_C| - |\texttt{nonMSS}_C|$. In fact, even a more general observation holds:

Definition 8.1 (wrapper and remainder)**.** *A set $\mathcal{W}$ of subsets of C is a* wrapper *iff* $\texttt{MSS}_C \subseteq \mathcal{W} \subseteq \texttt{SS}_C$*. Futhermore, the* remainder *of $\mathcal{W}$ is the set $\mathcal{R} = \mathcal{W} \cap \texttt{nonMSS}_C$.*

Proposition 8.1. *Let $\mathcal{W}$ be a wrapper and $\mathcal{R}$ its remainder. Then $|\texttt{MSS}_C| = |\mathcal{W}| - |\mathcal{R}|$.*

Proof. $\texttt{SS}_C = \texttt{MSS}_C \cup \texttt{nonMSS}_C$, and $\texttt{MSS}_C \cap \texttt{nonMSS}_C = \emptyset$, therefore $\texttt{MSS}_C = \mathcal{W} \backslash \texttt{nonMSS}_C = \mathcal{W} \backslash \mathcal{R}$, and since $\mathcal{R} \subseteq \mathcal{W}$, it holds $|\texttt{MSS}_C| = |\mathcal{W}| - |\mathcal{R}|$. □

Our approach for counting MSSes consists of the following steps. First, we find a *suitable* wrapper $\mathcal{W}$ and the corresponding remainder $\mathcal{R}$. Then, we encode the wrapper $\mathcal{W}$ and the corresponding remainder $\mathcal{R}$ with two formulas, $\mathbb{W}$ and $\mathbb{R}$, such that each *projected model* of $\mathbb{W}$ and $\mathbb{R}$ corresponds to an element of $\mathcal{W}$ and $\mathcal{R}$, respectively. Finally, we use a projected model counting tool to count the models of $\mathbb{W}$ and $\mathbb{R}$, and hence to determine $|\mathcal{W}|$ and $|\mathcal{R}|$ which yields also the MSS count $|\texttt{MSS}_C|$ (Proposition 8.1). In the following section, we provide details on *what is* and *how to find* a suitable wrapper, how to build the formulas $\mathbb{W}$ and $\mathbb{R}$, and what is the projection set.

8.3.2 Wrappers and Remainders

In this section, we gradually present 4 different wrappers $\mathcal{W}_1, \dots, \mathcal{W}_4$ and their corresponding remainders $\mathcal{R}_1, \dots, \mathcal{R}_4$. For each wrapper $\mathcal{W}_i$ and its remainder $\mathcal{R}_i$, we also build the corresponding formulas $\mathbb{W}_i$ and $\mathbb{R}_i$. Subsequently, we show how to combine multiple wrappers into a single, possibly more efficient, wrapper.

Wrapper $\mathcal{W}_1$ Our first wrapper $\mathcal{W}_1$ is simply the set $\texttt{SS}_C$ of all satisfiable subsets of C, and the corresponding remainder $\mathcal{R}_1$ is thus $\texttt{SS}_C \cap \texttt{nonMSS}_C = \texttt{nonMSS}_C$. We build the formula $\mathbb{W}_1$ using the variables $Vars(C)$ of C and an additional set of *activation variables* $A = \{a_f \,|\, f \in C\}$:

$$\mathbb{W}_1 = \bigwedge_{f \in C} (f \vee \neg a_f) \tag{8.1}$$

Note that the construction of *activation variables* as applied in formula $\mathbb{W}_1$ has been already used in the context of MSSes/MCSes/MUSes in various studies. For instance, we have used this construction in Chapter 7 in the QBF encoding of the *necessity problem* (Equation 7.11), or you can find it in the MUS-membership encoding by Janota and Marques-Silva [Janota and Marques-Silva, 2011].

Intuitively, by setting a variable a_f to *True*, we *activate* the subclause f in the clause $f' = (f \vee \neg a_f)$ of $\mathbb{W}_1$ since satisfying f is then the only way to satisfy f'. Let us denote by $AC(\pi)$ the one-to-one mapping (bijection) between a valuation π of A and the corresponding set of activated clauses of C, i.e., $AC(\pi) = \{f \in C \mid \pi(a_f) = \mathit{True}\}$.

Proposition 8.2. *For every valuation π of A, $\pi \in M_{\mathbb{W}_1\downarrow A}$ iff $AC(\pi) \in \mathcal{W}_1$. Consequently, $|M_{\mathbb{W}_1\downarrow A}| = |\mathcal{W}_1|$.*

Proof. $\Rightarrow$: Let π' be a model of $\mathbb{W}_1$ such that $\pi'_{\downarrow A} = \pi$. We show that $\pi' \models AC(\pi)$, hence $AC(\pi)$ is satisfiable and belongs to $\mathcal{W}_1$. For every $f \in AC(\pi)$, we know that $\pi' \models (f \vee \neg a_f)$. Moreover, by the definition of $AC(\pi)$, $\pi(a_f) = \mathit{True} = \pi'(a_f)$, i.e., $\pi' \not\models \neg a_f$, and thus $\pi' \models f$.
$\Leftarrow$: Let N be an element of $\mathcal{W}_1$ and ϕ its model. We define a valuation π' of $\mathbb{W}_1$ as $\pi'(x) = \phi(x)$ if $x \in \mathit{Vars}(C)$, $\pi'(a_f) = \mathit{False}$ if $f \notin N$, and $\pi'(a_f) = \mathit{True}$ if $f \in N$. Observe that $AC(\pi'_{\downarrow A}) = N$. Furthermore, $\pi' \models \mathbb{W}_1$: every clause $(f \vee \neg a_f) \in \mathbb{W}_1$ such that $f \in N$ is inherently satisfied by ϕ, and every clause $(f \vee \neg a_f) \in \mathbb{W}_1$ such that $f \notin N$ is satisfied by $\pi'(a_f) = \mathit{False}$. Hence, $\pi = \pi'_{\downarrow A} \in M_{\mathbb{W}_1\downarrow A}$ □

To build the formula $\mathbb{R}_1$ that encodes the remainder $\mathcal{R}_1 = \mathtt{SS}_C \cap \mathtt{nonMSS}_C$, i.e., the set of all satisfiable subsets of C that are not MSSes, we introduce another set B of *activation variables* $B = \{b_f \mid f \in C\}$. Similarly as in the case of A, given a valuation π of B, let us by $BC(\pi)$ denote the subset $\{f \in C \mid \pi(b_f) = \mathit{True}\}$ of C. By the definition of an MSS, a satisfiable subset N of C is not an MSS iff there exists a satisfiable N' such that $N \subsetneq N' \subseteq C$. We use $AC(\pi)$ and $BC(\pi)$ to encode such N and N', respectively, in $\mathbb{R}_1$:

$$\begin{aligned} \mathbb{R}_1 = \mathbb{W}_1 \wedge \bigwedge_{f' \in C'} (f' \vee \neg b_f) \wedge \bigwedge_{f \in C} (a_f \rightarrow b_f) \\ \wedge \bigvee_{f \in C} (\neg a_f \wedge b_f) \end{aligned} \tag{8.2}$$

Intuitively, the first conjunct $\mathbb{W}_1$ encodes that $AC(\pi)$ is satisfiable, the second conjunct encodes that $BC(\pi)$ is satisfiable, and the last two conjuncts express that $AC(\pi) \subsetneq BC(\pi)$. Note that since both the first conjuncts reason about satisfiability of subsets of C, we use in the second conjunct a primed version C' of C, i.e. a copy of C where each literal is substituted by its primed version.[2]

[2] One might note that since $AC(\pi) \subsetneq BC(\pi)$, then every model of $BC(\pi)$ is also a model of $AC(\pi)$, and thus we in fact do not need to introduce the primed copy C' of C. However, in the following, we compose wrapper $\mathcal{W}_1$ with other wrappers, and for this composition, we indeed need to introduce the primed copy.

Proposition 8.3. *For every valuation π of A, $\pi \in M_{\mathbb{R}_1\downarrow A}$ iff $AC(\pi) \in \mathcal{R}_1$. Consequently, $|M_{\mathbb{R}_1\downarrow A}| = |\mathcal{R}_1|$.*

Proof. $\Rightarrow$: Let π' be a model of $\mathbb{R}_1$ such that $\pi'_{\downarrow A} = \pi$. Since $\mathbb{R}_1$ subsumes $\mathbb{W}_1$, $\pi' \models \mathbb{W}_1$, and hence by Proposition 8.2 $AC(\pi)$ is satisfiable. The set $BC(\pi'_{\downarrow B})$ is defined and constrained in $\mathbb{R}_1$ analogously to $AC(\pi)$, thus $BC(\pi'_{\downarrow B})$ is also satisfiable. Furthermore, $AC(\pi) \subsetneq BC(\pi'_{\downarrow B})$ since $\pi' \models \bigwedge_{f \in C}(a_f \rightarrow b_f) \wedge \bigvee_{f \in C}(\neg a_f \wedge b_f)$.
$\Leftarrow$: Let N be an element of $\mathcal{R}_1$, N' a satisfiable superset of N, ϕ a model of N, and ϕ' a model of N'. We build a model π' of $\mathbb{R}_1$ as $\pi'(x) = \phi(x)$ if $x \in \mathit{Vars}(C)$, $\pi'(a_f) = \mathit{False}$ if $f \notin N$, $\pi'(a_f) = \mathit{True}$ if $f \in N$, $\pi'(x) = \phi'(x')$ if $x' \in \mathit{Vars}(C')$, $\pi'(b_f) = \mathit{False}$ if $f \notin N'$,

and $\pi'(b_f) = \mathit{True}$ if $f \in N'$. Analogously to the proof of Proposition 8.2, we know that $\pi' \models \mathbb{W}_1 \wedge \bigwedge_{f' \in C'} (f' \vee \neg b_f)$. As for the remaining part of $\mathbb{R}_1$, since $N \subsetneq N'$, we have that $\pi'(a_f) = \mathit{True}$ implies $\pi'(b_f) = \mathit{True}$ for every $f \in C$, and that there is at least one $f \in C$ such that $\pi'(a_f) = \mathit{False}$ and $\pi'(b_f) = \mathit{True}$. □

Based on the above observations, we can use a projected model counter to determine the cardinalities $|\mathcal{W}_1|$ and $|\mathcal{R}_1|$, and then employ Proposition 8.1 to deduce the MSS count. Thus, we are done... or are we not? The problem is the complexity (and practical tractability) of counting the projected models of $\mathbb{W}_1$ and $\mathbb{R}_1$. In general, there are two main criteria that affect the practical efficiency of the projected counting. One criterion is the cardinality of the projection set, which is in the case of $\mathcal{W}_1$ $|A| = |C|$. The other criterion is the number of the models, i.e., $|\mathcal{W}_1|$, which can be exponential w.r.t. $|C|$ since there are $2^{|C|}$ subsets of C and all of them (excluding the whole C) can be satisfiable. In the following, we propose three other wrappers (and corresponding remainders) that tend to optimize these two criteria by providing a better description of MSSes. All formulas that encode the following wrappers and their remainders use the same variables as in the case of $\mathcal{W}_1$ and $\mathcal{R}_1$, i.e., $\mathit{Vars}(C) \cup \mathit{Vars}(C') \cup A \cup B$. We also use the notation $AC(\pi)$ and $BC(\pi)$ to map valuations of A and B, respectively, to subsets of C.

Wrapper $\mathcal{W}_2$ Our second wrapper, $\mathcal{W}_2$, exploits the intersection $\mathtt{IMSS}_C$ of all MSSes of C. Clearly, for every MSS $N \in \mathtt{MSS}_C$ it holds that $\mathtt{IMSS}_C \subseteq N$. Thus, we could define the next wrapper as the set of all satisfiable subsets of C that are supersets of $\mathtt{IMSS}_C$. Unfortunately, computing the intersection can be very expensive (see below), thus, we exploit a more general observation:

Observation 8.1. *For every under-approximation I of $\mathtt{IMSS}_C$, i.e., $I \subseteq \mathtt{IMSS}_C$, and every MSS $N \in \mathtt{MSS}_C$, it holds that $I \subseteq N$.*

Given an under-approximation I of $\mathtt{IMSS}_C$, we define the second wrapper as $\mathcal{W}_2 = \{N \in \mathtt{SS}_C \mid I \subseteq N\}$. The formulas $\mathbb{W}_2$ and $\mathbb{R}_2$ that encode $\mathcal{W}_2$ and $\mathcal{R}_2$, respectively, are defined as follows:

$$\mathbb{W}_2 = \mathbb{W}_1 \wedge \bigwedge_{f \in I} a_f \tag{8.3}$$

$$\mathbb{R}_2 = \mathbb{W}_2 \wedge \mathbb{R}_1 \tag{8.4}$$

Proposition 8.4. *For every valuation π of A, $\pi \in M_{\mathbb{W}_2 \downarrow A}$ iff $AC(\pi) \in \mathcal{W}_2$. Consequently, $|M_{\mathbb{W}_2 \downarrow A}| = |\mathcal{W}_2|$.*

Proof. $\Rightarrow$: $\mathbb{W}_2$ subsumes $\mathbb{W}_1$, thus for every $\pi \in M_{\mathbb{W}_2 \downarrow A}$ the set $AC(\pi)$ is satisfiable (Proposition 8.2), and since $f \in AC(\pi)$ iff $\pi \models a_f$, $\bigwedge_{f \in I} a_f$ ensures that $I \subseteq AC(\pi)$.
$\Leftarrow$: Given $N \in \mathcal{W}_2$ and a model ϕ of N, we build a valuation π' of $\mathbb{W}_2$ as $\pi'(x) = \phi(x)$ if $x \in \mathit{Vars}(C)$, $\pi'(a_f) = \mathit{False}$ if $f \notin N$,

and $\pi'(a_f) = True$ if $f \in N$. Similarly as in the proof of Proposition 8.2, observe that $AC(\pi'_{\downarrow A}) = N$ and that $\pi' \models \mathbb{W}_2$, thus $\pi = \pi'_{\downarrow A} \in M_{\mathbb{W}_2 \downarrow A}$. □

Proposition 8.5. *For every valuation π of A, $\pi \in M_{\mathbb{R}_2 \downarrow A}$ iff $AC(\pi) \in \mathcal{R}_2$. Consequently, $|M_{\mathbb{R}_2 \downarrow A}| = |\mathcal{R}_2|$.*

Proof. $M_{\mathbb{R}_2 \downarrow A} = M_{(\mathbb{W}_2 \wedge \mathbb{R}_1) \downarrow A} = M_{\mathbb{W}_2 \downarrow A} \cap M_{\mathbb{R}_1 \downarrow A} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_2\} \cap \{\pi \,|\, AC(\pi) \in \mathcal{R}_1\} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_2 \cap \mathcal{R}_1\} = \{\pi \,|\, AC(\pi) \in \mathcal{R}_2\}$. The other direction holds since AC is a one-to-one mapping (bijection). □

Observe that by enforcing the variables $\{a_f \,|\, f \in I\}$ to be set to *True*, we effectively reduce the size of the projection set A since all models of $\mathbb{W}_2$ (and $\mathbb{R}_2$) agree on the assignment to these variables. In other words, $|M_{\mathbb{W}_2 \downarrow A}| = |M_{\mathbb{W}_2 \downarrow (A \setminus \{a_f \,|\, f \in I\})}|$ (and $|M_{\mathbb{R}_2 \downarrow A}| = |M_{\mathbb{R}_2 \downarrow (A \setminus \{a_f \,|\, f \in I\})}|$). Based on our practical experience, the intersection $\texttt{IMSS}_C$ is often relatively large, i.e., I can be also relatively large, and thus we can significantly reduce the projection set.

The remaining question is how to compute either exactly $\texttt{IMSS}_C$ or at least its under-approximation I. We are not aware of any work that would be dedicated to computing the intersection $\texttt{IMSS}_C$ of all MSSes of C. Yet, as shown in [Kullmann, 2000a], it holds that $\texttt{IMSS}_C = C \setminus \texttt{UMUS}_C$ where $\texttt{UMUS}_C$ is the union of all minimal unsatisfiable subsets (MUSes) of C. Recall that we have already discussed the computation of $\texttt{UMUS}_C$ in the previous chapter (Section 7.3.4). Based on a recent study [Mencía et al., 2019], computing $\texttt{UMUS}_C$, and hence also $\texttt{IMSS}_C$, is often practically intractable even for relatively small formulas. Yet, we can relatively cheaply compute a good over-approximation A of $\texttt{UMUS}_C$ whose complement, $C \setminus A$, is thus an under-approximation of $\texttt{IMSS}_C$. In the previous chapter, we presented an algorithm for computing an over-approximation of A: we first computed the lean kernel K of C to get an over-approximation of $\texttt{UMUS}_C$, and then we gradually refined the over-approximation by repeatedly solving a 2QBF formula. Here, we use only the lean kernel, since solving the 2QBF formula is too expensive in this case. In particular, we employ an approach from [Marques-Silva et al., 2014] to compute the lean kernel K and then use $I = C \setminus K$ to build the wrapper $\mathcal{W}_2$.

Similarly as we used the intersection $\texttt{IMSS}_C$, one might think of exploiting the union $\texttt{UMSS}_C$ of all MSSes; clearly, every MSS of C is contained in $\texttt{UMSS}_C$. Thus, at first glance, it makes sense to build a wrapper that contains all satisfiable subsets of C that are subsets of $\texttt{UMSS}_C$. Yet, we observe that $\texttt{UMSS}_C = C$ and thus the use of the union would not bring any benefit compared to $\mathcal{W}_1$:

Proposition 8.6. *For every formula C (with no empty clause) and the union $\texttt{UMSS}_C$ of all MSSes of C, it holds that $C = \texttt{UMSS}_C$.*

Proof. By contradiction, assume a clause $f \in C$ that is not contained in any MSS of C. Since f is non-empty, it is necessarily satisfiable, and hence either $\{f\}$ is an MSS of C or there exists an MSS N of C such that $N \supseteq \{f\}$. □

Wrapper $\mathcal{W}_3$ Our next wrapper, $\mathcal{W}_3$, is based on the following characterization of MSSes:

Proposition 8.7. *For every MSS N of C there exists a valuation ϕ of $Vars(C)$ such that $\phi \models N$ and for every $f \in C \backslash N$ it holds that $\phi \not\models f$.*

Proof. N is satisfiable, thus it has a model. For every model ϕ of N, if $\phi \models f$ for some $f \in C\backslash N$, then $\phi \models N \cup \{f\}$ which contradicts that N is an MSS. □

The corresponding wrapper for the property stated in Proposition 8.7 is $\mathcal{W}_3 = \{N \in \mathrm{SS}_C \,|\, \exists \phi \in Vals(C).\phi \models N \wedge \bigwedge_{f \in C\backslash N} \neg f\}$. The formulas $\mathbb{W}_3$ and $\mathbb{R}_3$ encoding $\mathcal{W}_3$ and $\mathcal{R}_3$, respectively, are the following:

$$\mathbb{W}_3 = \mathbb{W}_1 \wedge \bigwedge_{f \in C} (\neg a_f \rightarrow \neg f) \tag{8.5}$$

$$\mathbb{R}_3 = \mathbb{W}_3 \wedge \mathbb{R}_1 \tag{8.6}$$

Proposition 8.8. *For every valuation π of A, $\pi \in M_{\mathbb{W}_3 \downarrow A}$ iff $AC(\pi) \in \mathcal{W}_3$. Consequently, $|M_{\mathbb{W}_3\downarrow A}| = |\mathcal{W}_3|$.*

Proof. $\Rightarrow$: $\mathbb{W}_3$ subsumes $\mathbb{W}_1$, thus for every model π' of $\mathbb{W}_3$ such that $\pi = \pi'_{\downarrow A}$ it holds that $\pi' \models AC(\pi)$ (Proposition 8.2). Furthermore, by the definition of AC, $f \notin AC(\pi)$ iff $\pi \models \neg a_f$ (i.e., $\pi(a_f) = False$). Thus, the clauses $\bigwedge_{f \in C}(\neg a_f \rightarrow \neg f)$ of $\mathbb{W}_3$ ensure that for all $f \in C \backslash AC(\pi)$ it holds that $\pi' \not\models f$. Hence, π' is the model ϕ from the definition of $\mathcal{W}_3$.
$\Leftarrow$: Given $N \in \mathcal{W}_3$ and a model ϕ of $N \wedge \bigwedge_{f\in C\backslash N} \neg f$ (by the definition of $\mathcal{W}_3$), we build a valuation π' of $\mathbb{W}_3$ as $\pi'(x) = \phi(x)$ if $x \in Vars(C)$, $\pi'(a_f) = False$ if $f \notin N$, and $\pi'(a_f) = True$ if $f \in N$. Similarly as in the proof of Proposition 8.2, observe that $AC(\pi'_{\downarrow A}) = N$ and that $\pi' \models \mathbb{W}_3$, thus $\pi = \pi'_{\downarrow A} \in M_{\mathbb{W}_3\downarrow A}$. □

Proposition 8.9. *For every valuation π of A, $\pi \in M_{\mathbb{R}_3\downarrow A}$ iff $AC(\pi) \in \mathcal{R}_3$. Consequently, $|M_{\mathbb{R}_3\downarrow A}| = |\mathcal{R}_3|$.*

Proof. $M_{\mathbb{R}_3\downarrow A} = M_{(\mathbb{W}_3 \wedge \mathbb{R}_1)\downarrow A} = M_{\mathbb{W}_3\downarrow A} \cap M_{\mathbb{R}_1\downarrow A} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_3\} \cap \{\pi \,|\, AC(\pi) \in \mathcal{R}_1\} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_3 \cap \mathcal{R}_1\} = \{\pi \,|\, AC(\pi) \in \mathcal{R}_3\}$. The other direction holds since AC is a one-to-one mapping (bijection). □

Wrapper $\mathcal{W}_4$ Another wrapper, $\mathcal{W}_4$, is based on the following property of MSSes:

Proposition 8.10. *Let N be a maximal satisfiable subset of C and ϕ a valuation of $Vars(C)$. It holds that if $\phi \models N$, then $\phi \models \bigwedge_{f \in C\backslash N} \mathbb{P}$, where $\mathbb{P} = \bigwedge_{l \in f} \bigvee_{g \in \{g \in N \,|\, \neg l \in g\}} \bigwedge_{k \in g \backslash \{\neg l\}} \neg k$.*

Proof. By contradiction, assume a valuation ϕ such that $\phi \models N$ and $\phi \not\models \bigwedge_{f \in C \setminus N} \mathbb{P}$. Hence, there exists $f \in C \setminus N$ and $l \in f$ such that $\phi \not\models \bigvee_{g \in \{g \in N \,|\, \neg l \in g\}} \bigwedge_{k \in g \setminus \{\neg l\}} \neg k$. In other words, for every clause $g \in G = \{g \in N \,|\, \neg l \in g\}$ there is a literal $k \in g$, $k \neq \neg l$, with $\phi \models k$. Now, assume that we turn ϕ into a valuation ϕ' by only flipping the assignment to l, i.e., $\phi' \models l$. Clearly, $\phi' \models f$ since $l \in f$. Furthermore, $\phi' \models N \setminus G$ since these clauses do not contain $\neg l$ and thus the change of the assignment to l does not affect them. Finally, $\phi' \models G$ since every $g \in G$ contains a literal k, $k \neq \neg l$, with $\phi \models k$ (and ϕ' agrees with ϕ on k). Hence, $N \cup \{f\}$ is satisfiable, which contradicts that N is an MSS. $\square$

Informally, for every clause $f \in C \setminus N$ and every literal l of the clause, there is a clause $g \in N$ that *forces* l to be falsified. If there would be l that is not forced to be falsified, then the model ϕ can be relaxed to a model ϕ' that would satisfy $N \cup \{f\}$ (which is not possible since N is an MSS). Unfortunately, the proposition reasons about all models of an MSS (i.e., a universal property), which is expensive to encode with a propositional formula. Yet, since every MSS has at least a single model, we can relatively cheaply encode a weaker, existential, variant of Proposition 8.10. We define $\mathcal{W}_4$ as $\mathcal{W}_4 = \{N \in \mathrm{SS}_C \,|\, \exists \phi \in \mathit{Vals}(C). \phi \models N \wedge \bigwedge_{f \in C \setminus N} \mathbb{P}\}$, where $\mathbb{P} = \bigwedge_{l \in f} \bigvee_{g \in \{g \in N \,|\, \neg l \in g\}} \bigwedge_{k \in g \setminus \{\neg l\}} \neg k$. We encode $\mathcal{W}_4$ and its reminder $\mathcal{R}_4$ via $\mathbb{W}_4$ and $\mathbb{R}_4$ as follows:

$$\begin{aligned} \mathbb{W}_4 &= \mathbb{W}_1 \wedge \bigwedge_{f \in C} \neg a_f \rightarrow \mathbb{P}', \ \textit{where} \\ \mathbb{P}' &= \bigwedge_{l \in f} \bigvee_{g \in \{g \in C \,|\, \neg l \in g\}} (a_g \wedge \bigwedge_{k \in g \setminus \{\neg l\}} \neg k) \end{aligned} \tag{8.7}$$

$$\mathbb{R}_4 = \mathbb{W}_4 \wedge \mathbb{R}_1 \tag{8.8}$$

Proposition 8.11. *For every valuation π of A, $\pi \in M_{\mathbb{W}_4 \downarrow A}$ iff $AC(\pi) \in \mathcal{W}_4$. Consequently, $|M_{\mathbb{W}_4 \downarrow A}| = |\mathcal{W}_4|$.*

Proof. $\Rightarrow$: Let π' be a model of $\mathbb{W}_4$ such that $\pi = \pi'_{\downarrow A}$. We show that π' and $AC(\pi)$ comply with the conditions on ϕ and N, respectively, from the definition of $\mathcal{W}_4$. $\mathbb{W}_4$ subsumes $\mathbb{W}_1$, thus $\pi' \models AC(\pi)$ (Proposition 8.2). Furthermore, since $\pi' \models \mathbb{W}_4$ and $\pi = \pi'_{\downarrow A}$, it holds that $\pi' \models \mathbb{W}_4[\pi]$. Finally, as $f \in AC(\pi)$ iff $\pi \models a_f$, observe that $\mathbb{W}_4[\pi] \equiv \bigwedge_{f \in C \setminus AC(\pi)} \bigwedge_{l \in f} \bigvee_{g \in \{g \in AC(\pi) \,|\, \neg l \in g\}} \bigwedge_{k \in g \setminus \{\neg l\}} \neg k$ (which is the condition on ϕ).
$\Leftarrow$: Given $N \in \mathcal{W}_4$ and a valuation ϕ such that $\phi \models N \wedge \bigwedge_{f \in C \setminus N} \mathbb{P}$ (as in the definition of $\mathcal{W}_4$) , we build a model π' of $\mathbb{W}_4$ same as we did in the proof of Proposition 8.2, i.e., $\pi'(x) = \phi(x)$ if $x \in \mathit{Vars}(C)$, $\pi'(a_f) = \mathit{False}$ if $f \notin N$, and $\pi'(a_f) = \mathit{True}$ if $f \in N$. Like in the proof of Proposition 8.2, it holds that $AC(\pi'_{\downarrow A}) = N$ and $\pi' \models \mathbb{W}_1$. As for the rest of $\mathbb{W}_4$, it generally holds that $\pi' \models (\bigwedge_{f \in C} \neg a_f \rightarrow \mathbb{P}')$ iff $\pi' \models (\bigwedge_{f \in C} \neg a_f \rightarrow \mathbb{P}')[\pi'_{\downarrow A}]$. Furthermore, $(\bigwedge_{f \in C} \neg a_f \rightarrow \mathbb{P}')[\pi'_{\downarrow A}] \equiv$

$\bigwedge_{f \in C \setminus N} \mathbb{P}$, since $\pi'_{\downarrow A} \models a_f$ iff $f \in N$. Finally, $\phi \models \bigwedge_{f \in C \setminus N} \mathbb{P}$ and π' agrees with ϕ on $Vars(\bigwedge_{f \in C \setminus N} \mathbb{P})$, hence $\pi' \models \bigwedge_{f \in C \setminus N} \mathbb{P}$. □

Proposition 8.12. *For every valuation π of A, $\pi \in M_{\mathbb{R}_4 \downarrow A}$ iff $AC(\pi) \in \mathcal{R}_4$. Consequently, $|M_{\mathbb{R}_4 \downarrow A}| = |\mathcal{R}_4|$.*

Proof. $M_{\mathbb{R}_4 \downarrow A} = M_{(\mathbb{W}_4 \wedge \mathbb{R}_1) \downarrow A} = M_{\mathbb{W}_4 \downarrow A} \cap M_{\mathbb{R}_1 \downarrow A} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_4\} \cap \{\pi \,|\, AC(\pi) \in \mathcal{R}_1\} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_4 \cap \mathcal{R}_1\} = \{\pi \,|\, AC(\pi) \in \mathcal{R}_4\}$. The other direction holds since AC is a one-to-one mapping (bijection). □

8.3.3 Combining The Wrappers

Proposition 8.13. *For every two wrappers $\mathcal{W}_i, \mathcal{W}_j \in \{\mathcal{W}_1, \ldots, \mathcal{W}_4\}$ and their remainders $\mathcal{R}_i$, $\mathcal{R}_j$, it holds:*

1. *$\mathcal{W}_i \cap \mathcal{W}_j$ is a wrapper, and $\mathcal{R}_i \cap \mathcal{R}_j$ is its remainder.*
2. *For every valuation π of A, $\pi \in M_{(\mathbb{W}_i \wedge \mathbb{W}_j) \downarrow A}$ iff $AC(\pi) \in \mathcal{W}_i \cap \mathcal{W}_j$. Consequently, $|M_{(\mathbb{W}_i \wedge \mathbb{W}_j) \downarrow A}| = |\mathcal{W}_i \cap \mathcal{W}_j|$.*
3. *For every valuation π of A, $\pi \in M_{(\mathbb{R}_i \wedge \mathbb{R}_j) \downarrow A}$ iff $AC(\pi) \in \mathcal{R}_i \cap \mathcal{R}_j$. Consequently, $|M_{(\mathbb{R}_i \wedge \mathbb{R}_j) \downarrow A}| = |\mathcal{R}_i \cap \mathcal{R}_j|$.*

Proof.

1. By Definition 8.1, a set $\mathcal{W}$ is a wrapper iff $\texttt{MSS}_C \subseteq \mathcal{W} \subseteq \texttt{SS}_C$, and the remainder of $\mathcal{W}$ is $\mathcal{R} = \mathcal{W} \cap \texttt{nonMSS}_C$. $\mathcal{W}_i$ and $\mathcal{W}_j$ are wrappers, thus $\texttt{MSS}_C \subseteq \mathcal{W}_i, \mathcal{W}_j \subseteq \texttt{SS}_C$, and hence $\texttt{MSS}_C \subseteq \mathcal{W}_i \cap \mathcal{W}_j \subseteq \texttt{SS}_C$. Furthermore, $\mathcal{R}_i = \mathcal{W}_i \cap \texttt{nonMSS}_C$ and $\mathcal{R}_j = \mathcal{W}_j \cap \texttt{nonMSS}_C$, hence $\mathcal{R}_i \cap \mathcal{R}_j = \mathcal{W}_i \cap \mathcal{W}_j \cap \texttt{nonMSS}_C$.
2. By Propositions 8.2, 8.4, 8.8 and 8.11, $M_{(\mathbb{W}_i \wedge \mathbb{W}_j) \downarrow A} = M_{\mathbb{W}_i \downarrow A} \cap M_{\mathbb{W}_j \downarrow A} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_i\} \cap \{\pi \,|\, AC(\pi) \in \mathcal{W}_j\} = \{\pi \,|\, AC(\pi) \in \mathcal{W}_i \cap \mathcal{W}_j\}$. The other direction holds since AC is a one-to-one mapping (bijection).
3. By Propositions 8.3, 8.5, 8.9 and 8.12, $M_{(\mathbb{R}_i \wedge \mathbb{R}_j) \downarrow A} = M_{\mathbb{R}_i \downarrow A} \cap M_{\mathbb{R}_j \downarrow A} = \{\pi \,|\, AC(\pi) \in \mathcal{R}_i\} \cap \{\pi \,|\, AC(\pi) \in \mathcal{R}_j\} = \{\pi \,|\, AC(\pi) \in \mathcal{R}_i \cap \mathcal{R}_j\}$. The other direction holds since AC is a one-to-one mapping (bijection).

□

Note that all the formulas $\mathbb{W}_2$, $\mathbb{W}_3$ and $\mathbb{W}_4$ use as a subformula $\mathbb{W}_1$. Similarly, all the formulas $\mathbb{R}_2$, $\mathbb{R}_3$ and $\mathbb{R}_4$ use as a subformula $\mathbb{R}_1$. Thus, if we combine two wrappers, we duplicate some clauses in the formulas. In our implementation, we first remove all duplicated clauses from a formula before we pass it to a model counting tool. This simplification is sound as it does not reduce the number of models of the formula.

On Choice of Projected Model Counting

We remark on the choice of reduction of MSS counting to projected model counting. One might wonder whether we could have reduced to the classical problem of model counting. In this context, note that the classical model counting is #P-complete and checking whether an assignment satisfies a CNF formula is in P. In contrast, checking whether a given set of clauses is an MSS is in D^P, and therefore, it is expected to rely on a problem that is perhaps harder than classical model counting from the complexity perspective. Therefore, projected model counting, which is in #NP-hard, is a *good* choice given its hardness and the recent development of efficient techniques.

8.4 *Experimental Evaluation*

We have implemented our approach for solving the #MSS problem in a python-based tool. To count the number of projected models of the wrappers, we use the model counter GANAK [Sharma et al., 2019]. Furthermore, we employ the MaxSAT solver UWrMaxSat [Piotrów, 2019] as a backed while computing the under-approximation I of the intersection of MSSes in the case of the wrapper $\mathcal{W}_2$. Our tool is publicly available at:

https://github.com/jar-ben/MSSCounting

In this section, we experimentally compare the four wrappers, $\mathcal{W}_1, \ldots, \mathcal{W}_4$, and their combinations for the task of determining the MSS count of a given formula. Note that the wrapper $\mathcal{W}_1$ is by the definition subsumed by the remaining three wrappers. Therefore, there are only 8 possible combined wrappers (according to Proposition 8.13) that make sense: W1 $= \mathcal{W}_1$, W2 $= \mathcal{W}_2$, W3 $= \mathcal{W}_3$, W4 $= \mathcal{W}_4$, W23 $= \mathcal{W}_2 \cap \mathcal{W}_3$, W24 $= \mathcal{W}_2 \cap \mathcal{W}_4$, W34 $= \mathcal{W}_3 \cap \mathcal{W}_4$, W234 $= \mathcal{W}_2 \cap \mathcal{W}_3 \cap \mathcal{W}_4$. At first glance, the wrapper W234 that combines all the base wrappers should be the most suitable one since it provides the most accurate description of MSSes. However, the Boolean formula that describes this wrapper is also the largest one in the number of clauses, and thus it might be hard to deal with for the model counter. Therefore, it makes sense to evaluate all the 8 combinations. Moreover, we compare our wrapper-based approach for counting MSSes with the contemporary MSS counting approach: complete MSS enumeration via an MSS enumeration tool. In particular, we evaluate two contemporary MSS enumeration tools: FLINT [Narodytska et al., 2018][3], and RIME [Bendík and Černá, 2020b][4]. Thus, in total, we compare ten tools (RIME, FLINT, and our approach using one of the eight wrappers).

[3] The implementation of FLINT was kindly provided to us by its author, Nina Narodytska.

[4] https://github.com/jar-ben/rime

We use three comparison criteria: 1) the number of benchmarks for which the tools provide the MSS count, 2) the time to provide the MSS count, and 3) the scalability of the tools w.r.t. the MSS count.

We used a collection of 1200 Boolean CNF formulas that were recently used in prior MUS literature [Liu and Luo, 2018, Luo and Liu, 2019][5]. The benchmarks contain from 100 to 1000 clauses, use

[5] https://github.com/luojie-sklsde/MUS_Random_Benchmarks

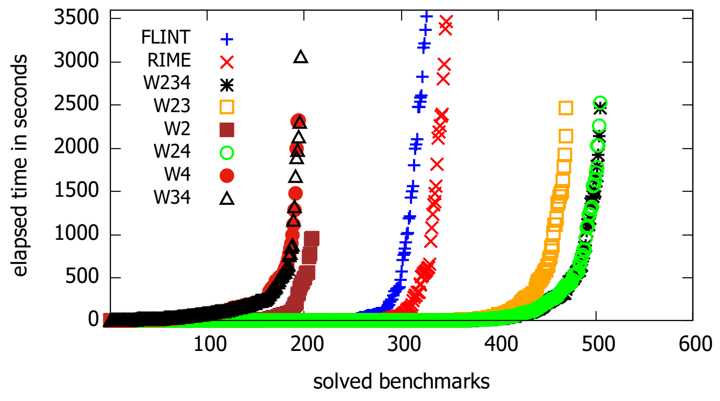


Figure 8.1: The number of solved benchmarks in time.

from 50 to 996 variables, and have from 2 to at least 4.74×10^{12} MSSes (the highest MSS count revealed in our evaluation).

All experiments were run using a time limit of 3600 seconds (1 hour) and computed on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux 4.19.67-2. Complete results are available in the online appendix[6].

[6] https://www.fi.muni.cz/~xbendik/phdThesis/

8.4.1 *Number of Completed Benchmarks*

We now examine the number of benchmarks for which the evaluated tools determined the MSS count; we simply say that the tools *solved* the benchmarks. Only 515 of the 1200 benchmarks were solved by at least one of the tools. Furthermore, 353 benchmarks were solved either by FLINT or by RIME, and 510 benchmarks were solved using one of our wrapper-based tools.

The cactus plot in Figure 8.1 shows for each tool the number of solved benchmarks and the time to solve the benchmarks. In particular, a point with coordinates $[x, y]$ means that there are x benchmarks for which the corresponding tool provided the MSS count within the first y seconds of the computation. There are only 327 and 347 benchmarks where FLINT and RIME provided the MSS count, respectively. As for our approach, the best result was achieved by the wrappers W234 and W24 which both solved 506 benchmarks, i.e., there is an incredible improvement of 46 percent over the best MSS enumerator RIME. W23 solved 471 benchmarks, which is also a solid result. On the other hand, W2, W4, and W34 solved only 209, 195, and 197 benchmarks, respectively. W1 and W3 did not solve even a single benchmark.

	FLINT	RIME	W234	W24	W23
FLINT	|	6;26	2;181	2;181	2;146
RIME	26;6	|	5;164	5;164	5;129
W234	181;2	164;5	|	4;4	35;0
W24	181;2	164;5	4;4	|	38;3
W23	146;2	129;5	0;35	3;38	|

Table 8.1: Number of benchmarks where a tool solved more;fewer benchmarks than the other tools.

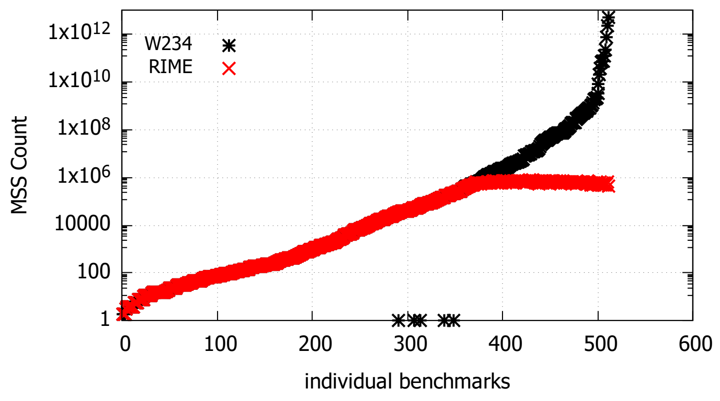


Figure 8.2: Scalability w.r.t. the MSS count.

There are 4 benchmarks that were solved by W234, but not by W24, and vice versa. Table 8.1 pair-wise compares FLINT, RIME, and the three best wrappers, W234, W24, and W23, w.r.t. this criterion. Each cell contains two numbers, k;l, expressing that there are k benchmarks that were solved by the tool labeling the row but not by the tool labeling the column, and vice versa for l. There are only 2 and 5 benchmarks that were solved by FLINT and RIME, respectively, and that were not solved by any of our wrappers.

8.4.2 *Scalability W.R.T. the MSS Count*

An MSS enumerator (e.g., FLINT or RIME) has to explicitly enumerate all MSSes to obtain the MSS count. Consequently, if the MSS count is large, the complete enumeration naturally becomes practically intractable (w.r.t. a reasonable time limit). On the other hand, our approach reduces the MSS counting to model counting, and it is often the case that a model counter needs to explicitly identify only a fraction of the models. Consequently, our approach should hypothetically scale much better w.r.t. the MSS count.

To prove our hypothesis, we compare the best MSS enumeration tool, RIME, with our best wrapper, W234. The plot in Figure 8.2 shows on the x-axis the benchmarks that were completed by at least one of the two tools and on the y-axis the MSS count of the benchmarks. The benchmarks are sorted by the MSS count. In the case of RIME, the points in the plot show the number of enumerated MSSes within the given time limit, i.e., it is either the exact MSS count or its under-approximation. RIME was able to solve only benchmarks with at most 10^6 MSSes. On the other hand, W234 solved even benchmarks that contain 10^{12} MSSes, i.e., it scales much better. Note that we show in the plot also the 5 benchmarks that were solved by RIME but not by W234; these are illustrated as the 5 points on the x-axis.

8.5 *Summary and Future Work*

Motivated by the progress in model counting, we initiate the study of counting the number of MSSes of a given formula. Our novel algorithmic framework relies on the notions of wrappers and their corresponding remainders. We show that wrappers and remainders compose, and the computation of the sizes of wrappers and remainders reduces to the projected model counting. The availability of an efficient projected model counter, GANAK, allowed our MSS counting approach to scale, in terms of the MSS count, significantly better than alternative approaches based on the MSS enumeration.

As for the future work, we would like to address also a better scaling of our approach w.r.t. the number of clauses in the input instance. Whereas we are currently able to handle instances with hundreds of clauses, instances with high thousands or even millions of clauses are still out of our reach. In this context, a promising challenge would be to handle the widely used dataset of 291 CNF formulas from the MUS track of the SAT Competition 2011. A vast majority of benchmarks from this set is not tractable for contemporary MSS enumeration tools due to a large number of MSSes, and it is also not tractable for our approach owing to a large number of clauses in the benchmarks, which in turn leads to an increase in the number of variables for projected model counting queries. An interesting direction to address this scalability challenge is to investigate whether a component caching-based scheme operating natively over the space of MSSes, i.e., avoiding the reduction to model counting, can lead to a better runtime efficiency.

Part IV

Minimal Inductive Validity Cores

9
Minimal Inductive Validity Cores

In this chapter, we focus on another instance of minimal sets over a monotone predicate called *Minimal Inductive Validity Cores (MIVCs)*, which find application in the area of symbolic model checking. In particular, we present an algorithm for enumeration of MIVCs, called GROW-SHRINK [Bendík et al., 2018c].

Symbolic model checking using induction-based techniques such as IC3/PDR [Eén et al., 2011], *k*-induction [Sheeran et al., 2000], and *k*-liveness [Claessen and Sörensson, 2012] can be used to determine whether properties hold of complex finite or infinite-state systems. Such tools are popular both because they are highly automated (often requiring no user interaction other than the specification of the model and desired properties), and also because, in the event of a violation, the tool provides a counterexample demonstrating a situation in which the property fails to hold. These counterexamples can be used both to illustrate subtle errors in complex hardware and software designs [Murugesan et al., 2013, Miller et al., 2010] and to support automated test case generation [Whalen et al., 2013, You et al., 2015].

If a property is proved, however, most model checking tools do not provide additional information. This can lead to situations in which developers have an unwarranted level of confidence in the behavior of the system. Issues such as vacuity [Kupferman and Vardi, 2003], incorrect environmental assumptions [Whalen et al., 2007], and errors either in English language requirements or formalization can all lead to failures of "proved" systems. Thus, even if proofs are established, one must approach verification with skepticism.

A few years ago, *proof cores*[1] have been proposed as a mechanism to determine which elements of a model are used when constructing a proof. This idea is formalized by Ghassabani et al. for inductive model checkers as *Inductive Validity Cores* (IVCs) [Ghassabani et al., 2016]. IVCs offer proof explanation as to why a property is satisfied by a model in a formal and human-understandable way. The idea is to lift Boolean unsat cores [Zhang and Malik, 2003] to the level of sequential model checking algorithms using induction. Informally, if a model is viewed as a conjunction of constraints, a minimal IVC (MIVC) is a set of constraints that is sufficient to construct a proof

[1] `https://www.cadence.com/`

such that if any constraint is removed, the property is no longer valid. Depending on the model and property to be analyzed, there are many possible MIVCs, and there is often substantial diversity between the IVCs used for proof. In recent studies [Ghassabani et al., 2016, Murugesan et al., 2016, Ghassabani et al., 2017a,b], there have been explored several different uses of IVCs, including:

Traceability: Minimal inductive validity cores can provide accurate traceability matrices with no user effort. Given multiple MIVCs, *rich traceability* matrices [Murugesan et al., 2016] can be automatically constructed that provide additional insight about *required* vs. *optional* design elements.

Vacuity detection: Syntactic vacuity detection (checking whether all subformulae within a property are necessary for its validity) has been well studied [Kupferman and Vardi, 2003]. MIVCs allow a generalized notion of vacuity that can indicate weak or mis-specified properties even when a property is syntactically non-vacuous.

Coverage analysis: Coverage analysis provides a metric for testing whether a set of properties is adequate for the model. Several different notions of coverage have been proposed [Chockler et al., 2006, Kupferman et al., 2008], but these tend to be very expensive to compute. MIVCs provide an inexpensive coverage metric by determining the percentage of model atoms necessary for proofs of all properties.

Impact Analysis: Given a single (or for more accurate results, all) MIVC, it is possible to determine which requirements may be falsified by changes to the model. This analysis allows for selective regression verification of tests and proofs: if there are alternate proof paths that do not require the modified portions of the model, then the requirement does not need to be re-verified.

Design Optimization: A practical way of calculating all MIVCs allows synthesis tools to find a minimum set of design elements (optimal implementation) for a certain behavior. Such optimizations can be performed at different levels of synthesis.

To be useful for these tasks, the generation process must be efficient and the generated IVC must be accurate and precise (that is, sound and minimal). In their previous work, Ghassabani et al. developed an efficient *offline* algorithm [Ghassabani et al., 2017b] for finding all minimal IVCs based on the MARCO algorithm for MUSes [Liffiton et al., 2016]. The algorithm is considered *offline* because it during its computation produces both minimal and non-minimal IVCs, however, the information about the minimality of the provided solutions is not determined until all MIVCs have been computed. In cases in which models contain many MIVCs, this approach can be impractically expensive or simply not terminate within a reasonable time limit.

In this chapter, we present our algorithm for *online* MIVC enumeration, called GROW-SHRINK [Bendík et al., 2018c]. With this algorithm, solutions are produced incrementally, and each solution produced is guaranteed to be minimal. Therefore, the algorithm produces at least some MIVCs even in the case of models for which a complete MIVC enumeration is intractable. Moreover, the proposed algorithm is often more efficient then the baseline offline algorithm also in the case of tractable models. We demonstrate this via an experimental evaluation.

The rest of the chapter is organized as follows. In Section 9.1 we define all the necessary notions. Section 9.2 summarizes the existing techniques. In Section 9.3 we present our novel algorithm. Section 9.5 provides an example execution of our algorithm. Finally, sections 9.4 and 9.6 cover implementation details and present experimental results.

9.1 *Preliminaries*

A transition system (I, T) over a state space S consists of an initial state predicate $I : S \rightarrow \{True, False\}$ and a transition step predicate $T : S \times S \rightarrow \{True, False\}$. The notion of reachability for (I, T) is defined as the smallest predicate $R : S \rightarrow \{True, False\}$ satisfying the following formulae:

$$\forall s \in S : \ I(s) \rightarrow R(s)$$

$$\forall s, s' \in S : \ R(s) \wedge T(s, s') \rightarrow R(s')$$

A safety property $P : S \rightarrow \{True, False\}$ holds on a transition system (I, T) iff it holds on all reachable states, i.e., $\forall s \in S : \ R(s) \rightarrow P(s)$. We denote this by $(I, T) \vdash P$. We assume the transiton step predicate T is equivalent to a conjunction $t_1 \wedge \cdots \wedge t_n$ of transition step predicates $t_1, \ldots, t_n$, called top level conjuncts[2]. In such case, T can be identified with the set of its top level conjuncts $\{t_1, \ldots, t_n\}$. By further abuse of notation, we write $T \backslash \{t\}$ to denote the removal of a top level conjunct t from T, and $T \cup \{t\}$ to denote the addition of a top level conjunct t to T.

[2] Note that this is usually the case when (I, T) represents a synchronous system, i.e., a system where all components (variables) of the system change state synchronously during each tick of the system clock. The individual top-level conjuncts constraint the behavior of individual components (variables) of the system.

Definition 9.1 (IVC, MIVC)**.** *A set of conjuncts $U \subseteq T$ is an* Inductive Validity Core (IVC) *for $(I, T) \vdash P$ iff $(I, U) \vdash P$. Moreover, U is a* Minimal IVC (MIVC) *for $(I, T) \vdash P$ iff $(I, U) \vdash P$ and $\forall t \in U : \ (I, U \backslash \{t\}) \nvdash P$.*

Definition 9.2 (inadequacy, MIS)**.** *A set of conjuncts $U \subseteq T$ is an* inadequate *set for $(I, T) \vdash P$ iff $(I, U) \nvdash P$. Especially, $U \subseteq T$ is a* Maximal Inadequate Set (MIS) *for $(I, T) \vdash P$ iff U is inadequate and $\forall t \in (T \backslash U) : \ (I, U \cup \{t\}) \vdash P$.*

Inadequate sets are duals to inductive validity cores. Each $U \subseteq T$ is either inadequate set or an inductive validity core. In order to unify the notation, we use notation *inadequate* and *adequate*. Note that especially minimal inductive validity cores can be thus called minimal adequate sets. Furthermore, since we use (I, T) to denote the input transition system and P to denote the safety property of interest throughout the whole chapter, we slightly abuse the notation and simply write that "a subset U of T is an MIVC" instead of the full term "a subset U of T is an MIVC for $(I, T) \vdash P$". Similarly, we simply write that a subset U of T "is adequate/inadequate" instead of "is adequate/inadequate for $(I, T) \vdash P$".

Proposition 9.1 (Monotonicity). *If a set of conjuncts $U \subseteq T$ is adequate then all its supersets are adequate as well:*

$$\forall U_1 \subseteq U_2 \subseteq T : (I, U_1) \vdash P \Rightarrow (I, U_2) \vdash P.$$

Symmetrically, if $U \subseteq T$ is inadequate then all its subsets are inadequate as well:

$$\forall U_1 \subseteq U_2 \subseteq T : (I, U_2) \nvdash P \Rightarrow (I, U_1) \nvdash P.$$

Proof. If $U_1 \subseteq U_2$ then reachable states of (I, U_2) form a subset of the reachable states of (I, U_1). □

Observation 9.1. *Minimal Inductive Validity Cores (MIVCs) are an instance of Minimal Sets over a Monotone Predicate (MSMPs) as defined in Section 2.2. In particular, the set C of elements is the set T of predicates and for every subset N of T the predicate $\mathbf{P}$ is defined as $\mathbf{P}(N) = 1$ iff N is adequate. The monotonicity of $\mathbf{P}$ is witnessed in Proposition 9.1. Hence, $\mathbf{P}_1$-minimal subsets correspond to MIVCs and $\mathbf{P}_0$-maximal subsets correspond to MISes.*

Since MIVCs are instance of MSMPs, we can adopt the concept of unexplored subsets as defined in Section 2.3.1, including the definitions of maximal and minimal unexplored subsets. We keep the notation of `Unexplored` to denote the set of all unexplored subsets, and we employ the symbolic technique based on the formula $map^+ \wedge map^-$ to maintain the set `Unexplored` (described Section 2.3.2). We also adopt the concept of *critical elements*, however, here we call them *critical predicates*. In particular, a predicate $t \in N$ is critical for an adequate N iff $N \backslash \{t\}$ is inadequate.

Finally, we adopt the terminology of shrinking and growing. In particular, a shrinking procedure shrinks a given adequate subset N of T into an MIVC. Similarly, a growing procedure grows a given inadequate subset M of T into an MIS M' such that $M \subseteq M' \subseteq T$.

Example 9.1. *We adopt an example from [Ghassabani et al., 2017b] to illustrate the concept of MIVCs on a simple synchronous system from the avionics domain. An Altitude Switch (ASW) is a control device that turns power on and off to the Device of Interest (DOI) based on the altitude of the aircraft. When the aircraft descends below a threshold value, ASW turns the power to DOI on, and when the aircraft ascends over the threshold plus some hysteresis factor, ASW again turns the power to DOI off.*

In Algorithm 9.1, we show an implementation of ASW containing two altimeters written in the Lustre language [Halbwachs et al., 1991] (a language for modeling synchronous systems). If an altimeter is below the `THRESHOLD` value, the DOI is turned on; otherwise, if the system is inhibited or both the altimeters are above the threshold value plus a hysteresis factor `T_HYST`, then the DOI is turned off; if neither of the two conditions holds, then DOI is initially turned off and thereafter it retains its previous value. Namely, "`(false - > pre(doi_on))`" in equation (7) describes an

```
node asw (alt1, alt2: int) returns (doi_on: bool);
    var
      a1_below, a2_below, a1_above, a2_above,
      one_below, both_above, on_p: bool;
    let
(1)     a1_below = (alt1 < THRESHOLD);
(2)     a2_below = (alt2 < THRESHOLD);
(3)     a1_above = (alt1 ⩾ T_HYST);
(4)     a2_above = (alt2 ⩾ T_HYST);
(5)     one_below = a1_below or a2_below;
(6)     both_above = a1_above and a2_above;
(7)     doi_on = if one_below then true
                 else if both_above then false
                 else (false − > pre(doi_on));
(8)     on_p = ((alt1 < THRESHOLD) and
               (alt2 < THRESHOLD)) ⇒ doi_on;
    tel;
```

Algorithm 9.1: Lustre implementation of ASW from Example 9.1.

initialization of a register: in the first step (tick of the system clock), the expression is `false`*, and thereafter it retains the previous value of* `doi_on`*. The property of interest* `on_p`*, defined in equation (8), expresses that if both altimeters are under the threshold, then the DOI is turned on. This property holds in the model and it can be proved e.g. via a k-induction verifier such as* `JKind` *[Gacek et al., 2018].*

If we encode the Lustre program as a transition system, then each of the eight equations of the program corresponds to a transition step predicate; for simplicity, we name the predicates after the eight variables, i.e., $T = \{$`a1_below`*,* `a2_below`*,* `a1_above`*,* `a2_above`*,* `one_below`*,* `both_above`*,* `doi_on`*,* `on_p`*$\}$. There are two MIVCs for the transition system and the property of interest* `on_p`*: $\{$*`a1_below`*,* `one_below`*,* `doi_on`*,* `on_p`*$\}$ and $\{$*`a2_below`*,* `one_below`*,* `doi_on`*,* `on_p`*$\}$. This is because in the implementation, the DOI is turned on when at least one of the altimeters is below the threshold value, while the property of interest states that they both must be below. Note that there are three transition step predicates, $\{$*`a1_above`*,* `a2_above`*,* `both_above`*$\}$, that do not appear in any of the MIVCs. Consequently, the three predicates are not necessary for the satisfaction of the property.*

9.2 *Related Work*

Ghassabani et al. [Ghassabani et al., 2017b] proposed an *online* algorithm for MIVC enumeration which is based on the MUS enumeration algorithm MARCO [Liffiton et al., 2016]. The algorithm iteratively chooses maximal unexplored subsets and tests them for adequacy. Each maximal subset that is found to be adequate is then shrunk into an MIVC via a custom shrinking procedure. This algorithm enumerates MIVCs in an online manner with a relatively steady rate of the enumeration. However, the authors of the algorithm found it to be rather slow since the shrinking procedure can be extremely time consuming as each check for adequacy is in fact a model checking problem.

Therefore, Ghassabani et al. [Ghassabani et al., 2017b] proposed another algorithm which, instead of computing MIVCs in on online manner, rather computes only *approximately* minimal IVCs. The algorithm is referred to as *offline algorithm*. In particular, the algorithm iteratively picks maximal unexplored subsets, checks them for adequacy, and turns the adequate subsets into approximately minimal IVCs using an approximate shrinking algorithm called `IVC_UC` [Ghassabani et al., 2016]. `IVC_UC` is able to identify IVCs which are often very close to actual MIVCs, yet cheap to compute. The offline algorithm does not provide any guarantee about the minimality of the gradually provided IVCs until all subsets of T become explored. At the end of the computation, the minimal and non-minimal produced IVCs are distinguished using the subset inclusion relation. An experimental evaluation shows that the *offline* algorithm computes all MIVCs much faster than the *online* algorithm. However, it does not enumerate MIVCs online and thus on benchmarks that contain a large number of MIVCs may produce no (guaranteed) MIVCs within a given time limit.

9.3 *Algorithm*

In this section, we present our algorithm for online MIVC enumeration, called GROW-SHRINK [Bendík et al., 2018c]. As the name of the algorithm suggests, it is based on a growing and a shrinking procedure. In the following, we first describe in Sections 9.3.1 and 9.3.2 the purpose of the two procedures and we sketch how they work. Subsequently, in Section 9.3.4, we combine the two procedures into the overall MIVC enumeration algorithm (GROW-SHRINK).

9.3.1 *Shrinking Procedure*

The purpose of the shrinking procedure is to identify individual minimal IVCs. The base workflow of the shrinking procedure is adopted from the domain agnostic single MSMP extractor we presented in Section 2.3.3, Algorithm 2.1. The input of the shrinking procedure is an adequate subset U of T such that all adequate *proper* subsets of U are unexplored[3]. The output is an MIVC U_{mivc} such that $U_{mivc} \subseteq U$.

[3] Note that this is a departure from previous chapters where we assumed that the input set U that we shrink is unexplored; here, due to technical reasons, we only require proper subsets of U to be unexplored.

The shrinking procedure is shown in Algorithm 9.2. The algorithm iteratively maintains two sets: the input set U and a set K of predicates that are critical for U. Initially, $K = \emptyset$. In each iteration, the algorithm picks a predicate $t \in U \setminus K$ and checks if $U \setminus \{t\}$ is inadequate. If it is the case, then t is critical for U and hence it is added to K. Otherwise, if $U \setminus \{t\}$ is adequate, the predicate t is removed from U. The check for the adequacy of $U \setminus \{t\}$ is performed in

```
input  : an adequate set $U$
output: an MIVC $U_{mivc}$ such that $U_{mivc} \subseteq U$
1 $K = \emptyset$
2 while $U \neq K$ do
3     $t \leftarrow$ pick from $U \backslash K$
4     if $U \backslash \{t\} \notin$ Unexplored then $K \leftarrow K \cup \{t\}$
5     else
6         if isAdequate($U \backslash \{t\}$) then $U \leftarrow U \backslash \{t\}$
7         else $K \leftarrow K \cup \{t\}$
8 return $U$
```

Algorithm 9.2: A base single MIVC extraction procedure (i.e., shrinking) of GROW-SHRINK.

two steps. First, the algorithm tests whether $U\backslash\{t\}$ is explored and if yes, then $U\backslash\{t\}$ is necessarily inadequate since we assume that all adequate proper subsets of U are unexplored. Second, if $U\backslash\{t\}$ is unexplored, then it is checked for adequacy with a model checker, which we denote by isAdequate($U\backslash\{t\}$).

Proposition 9.2. *Given an adequate subset U_{init} of T such that all adequate proper subsets of U_{init} are unexplored, Algorithm 9.2 returns an MIVC U_{mivc} such that $U_{mivc} \subseteq U_{init}$. Moreover, since all adequate proper subsets of U_{init} are unexplored, then $U_{mivc} = U_{init}$ or U_{mivc} is unexplored.*

Proof. The invariant of the algorithm is that $K \subseteq U$ and every $t \in K$ is critical for U. In every iteration, either K is enlarged or U is reduced, hence eventually $U = K$ and the algorithm terminates. At this point, since every $t \in U$ is critical for U, then U is an MIVC (Observation 2.7). □

The most expensive part of the procedure are the model checking calls. In the worst case, the algorithm performs k model checking calls where k is the number of predicates in the input set U. In the best case, the input U is an MIVC and for every $t \in U$ the set $U\backslash\{t\}$ is explored, hence no model checking call is performed. Crucially, observe that it is the explored *inadequate* sets that allows us to save model checking calls during shrinking. Consequently, it might be worth to explore (at least some) inadequate sets before the shrinking. And that is the purpose of the growing procedure: to explore many inadequate subsets, which in turn boosts the shrinking.

9.3.2 *Growing Procedure*

Recall that if a set N is determined to be inadequate then all of its $2^{|N|}$ subsets are necessarily also inadequate. Therefore, the larger is the set that is determined to be inadequate, the more inadequate sets are explored. Thus, at the first glance, to identify inadequate sets as quickly as possible, we should search for maximal inadequate subsets (MISes) of T.

To find an MIS, we can identify an inadequate subset U of T and then grow it to an MIS U_{mis} such that $U \subseteq U_{mis} \subseteq T$. To implement

```
input  : inadequate $U$
output: approximately maximal inadequate set
1 $M \leftarrow$ a maximal $M \in$ Unexplored such that $M \supseteq U$
2 while isAdequate($M$) do
3     $M_{ivc} \leftarrow$ IVC_UC($M$)                    // gets approximately minimal IVC
4     $t \leftarrow$ pick a predicate from $(M_{ivc} \backslash U)$
5     $M \leftarrow M \backslash \{t\}$
6 return $M$
```

Algorithm 9.3: A base approximate single MIS extraction procedure (i.e., growing) of GROW-SHRINK.

the grow, we can dualize the shrinking procedure presented in the previous section. In particular, we can iteratively attempt to add elements from $T \backslash U$ to U, checking each new set for adequacy and keeping only changes that leave the set inadequate. Same as in the case of the shrinking procedure, we can use the set `Unexplored` to avoid checking sets whose status is already known. However, in the worst case, such procedure can still perform $|T \backslash U|$ model checking calls, which is quite expensive.

Instead, we propose to use a different approach. Algorithm 9.3 shows a procedure that, given an inadequate subset U of T, finds an *approximately* maximal inadequate set. It first picks a maximal unexplored set M such that $M \supseteq U$ and checks it for adequacy. If M is inadequate, then it is necessarily an MIS (Observation 2.12). Otherwise, if M is adequate then it is iteratively reduced until an inadequate set is found. In particular, whenever M is found to be adequate, the approximative algorithm IVC_UC [Ghassabani et al., 2016] is used to find an approximately minimal IVC M_{ivc} of M. M_{ivc} succinctly explains M's adequacy. In order to turn M into an inadequate set, it is reduced by one element from $M_{ivc} \backslash U$ and checked for adequacy. If M is still adequate then the approximate growing procedure continues with a next iteration. Otherwise, if M is inadequate, the procedure finishes.

Proposition 9.3. *Given an unexplored inadequate subset U of T, Algorithm 9.3 returns an* unexplored *inadequate subset M of T.*

Proof. Let us denote initial M as M_{init}. Since $M_{init} \supseteq U$ and M is recursively reduced only by elements that are not contained in U, then in every iteration it holds that $U \subseteq M \subseteq M_{init}$. Since both U, M_{init} are unexplored, then M is necessarily also unexplored. □

9.3.3 *Procedure* isAdequate

In Algorithm 9.4, we describe the procedure isAdequate that checks whether a given subset of elements $U \subseteq T$ is adequate, i.e., it checks whether U is sufficient to prove the property of interest P. Note that performing such a check is as hard as model checking ([Ghassabani et al., 2016], Theorem 1). Thus, in the general case, determining whether a set of model elements is an MIVC may not be possible

```
1 $res \leftarrow$ checkAdq($U$)
2 if $res$ = UNKNOWN then
3     $approximateWarning \leftarrow true$                    // a global variable
4 return ($res$ = ADEQUATE)
```

Algorithm 9.4: isAdequate(U)

for model checking problems that are in general undecidable, such as those involving infinite theories. We assume there is a function checkAdq that checks whether or not P is provable for the transition system (I, U). The function checkAdq can return UNKNOWN (after a user-defined timeout) as well as ADEQUATE or INADEQUATE. For a given set U, if our implementation is unable to prove the property, we conservatively assume that the property is falsifiable (i.e. U is inadequate) and we set a global warning flag *approximateWarning* to the user that the results produced may be approximate.

9.3.4 *Complete Algorithm*

In this section, we describe how to combine the shrink and grow methods to form an efficient online MIVC enumeration algorithm. We call the algorithm GROW-SHRINK. Since knowledge of (approximately) maximal inadequate subsets can be exploited to speed up the shrinking procedure, it might be tempting to first find all MISes. However, this is in general intractable since there can be up to exponentially many MISes (w.r.t. $|T|$). Instead, we propose to alternate both the shrinking and growing procedures. Note that during shrinking, we might determine some subsets to be inadequate. Such subsets can be subsequently used as *seeds* for growing. Dually, adequate subsets that are explored during growing can be later used as *seeds* for the shrinking procedure.

The pseudocode of our algorithm is shown in Algorithm 9.5. The computation of the algorithm starts with an initialisation procedure init which creates a global variable `Unexplored` for maintaining the unexplored subsets and a global shrinking queue *shrinkingQueue* for storing seeds for the shrinking procedure. Then the main procedure findMIVCs of our algorithm is called.

Procedure findMIVCs works iteratively. In each iteration, the procedure picks a maximal unexplored subset U_{max} and checks it for adequacy. If U_{max} is inadequate, then U_{max} and all of its subsets are marked as explored. Otherwise, if U_{max} is adequate, then the algorithm IVC_UC [Ghassabani et al., 2016] is used to reduce U_{max} into an approximately minimal IVC, and subsequently the procedure `Shrink` is used to shrink it into an MIVC.

Procedure shrink works as described in Section 9.3.1. However, besides shrinking the given set into an MIVC, the procedure has also another purpose. Every inadequate set that is found during the shrinking is stored in a queue *growingQueue*. At the end of the

```
Function init($(I, T), P$):
    input : a transition system $(I, T)$
    input : a safety property $P$ such that $(I, T) \vdash P$
    output: all MIVCs for $(I, T) \vdash P$
    Unexplored ← $\mathcal{P}(T)$                              // a global variable
    shrinkingQueue ← empty queue                               // a global variable
    approximateWarning ← false                                 // a global variable
    findMIVCs()
Function findMIVCs():
    while Unexplored $\neq \emptyset$ do
        $U_{max}$ ← a maximal set ∈ Unexplored
        if isAdequate($U_{max}$) then
            $U_{ivc}$ ← IVC_UC($U_{max}$)
            shrink($U_{ivc}$)
        else
            Unexplored ← Unexplored$\setminus\{N \mid N \subseteq U_{max}\}$
        while shrinkingQueue is not empty do
            $U$ ← dequeue(shrinkingQueue)
            shrink($U$)
Function shrink($U$):
    $K \leftarrow \emptyset$
    growingQueue ← empty queue
    while $U \neq K$ do
        $t$ ← choose $t \in U\setminus K$
        if $U\setminus\{t\} \notin$ Unexplored then $K \leftarrow K \cup \{t\}$
        else
            if isAdequate($U\setminus\{t\}$) then $U \leftarrow U\setminus\{t\}$
            else
                $K \leftarrow K \cup \{t\}$
                enqueue(growingQueue, $U\setminus\{t\}$)
    output $U$                                                 // Output Minimal IVC
    updateShrinkingQueue($U$)
    Unexplored ← Unexplored$\setminus\{N \mid N \supseteq U\}$
    while growingQueue is not empty do
        $V$ ← dequeue(growingQueue)
        grow($V$)
Function grow($V$):
    $M$ ← a maximal set ∈ Unexplored such that $M \supseteq V$
    while isAdequate($M$) do
        $M_{ivc}$ ← IVC_UC($M$)
        updateShrinkingQueue($M_{ivc}$)
        enqueue(shrinkingQueue, $M_{ivc}$)
        Unexplored ← Unexplored$\setminus\{N \mid N \supseteq M_{ivc}\}$
        $t$ ← choose $t \in (M_{ivc}\setminus V)$
        $M \leftarrow M\setminus\{t\}$
    Unexplored ← Unexplored$\setminus\{N \mid N \subseteq M\}$
Function updateShrinkingQueue($U$):
    for $V \in$ shrinkingQueue do
        if $U \subseteq V$ then remove $V$ from shrinkingQueue
```

Algorithm 9.5: The GROW-SHRINK algorithm.

procedure, all of these inadequate sets are grown into approximately maximal inadequate sets using the procedure grow.[4]

[4] Note that one might think of improving the shrinking procedure via the MIVC approximation algorithm IVC_UC. In particular, every time we find $U\backslash\{t\}$ to be adequate, one could use IVC_UC to compute an approximatively minimal IVC A of $U\backslash\{t\}$ and then reduce U to A instead of reducing U just to $U\backslash\{t\}$. Such a step would be equivalent to the usage of unsat cores during shrinking in the Boolean CNF domain. The thing is that the execution of IVC_UC is not so cheap as the extraction of unsat cores. Moreover, based on our empirical experience, the usage of IVC_UC during shrinking usually do not significantly reduce U since then initial U is often very close to an MIVC (due to the single call of IVC_UC before the shrinking).

Procedure grow turns a given inadequate set V into an approximately maximal inadequate set M as described in Section 9.3.2. The resultant set and all of its subsets are marked as explored. Moreover, every adequate set found during the growing is marked as explored and enqueued into *shrinkingQueue*. The queue *shrinkingQueue* is dequeued at the end of each iteration of the main procedure findMIVCs and the sets that were stored in the queue are shrunk to MIVCs.

Finally, note that every time an MIVC is found and every time a set is added to *shrinkingQueue*, GROW-SHRINK calls the procedure UpdateShrinkingQueue which, given an adequate set U, removes from *shrinkingQueue* all supersets of U. Due to this procedure, the algorithm maintains the following invariants about *shrinkingQueue*:

I1) For each already produced MIVC X it holds that there is no U in the queue such that $X \subseteq U$.

I2) There are no two X, U in the queue such that $X \subseteq U$.

Correctness Clearly, the algorithm terminates and all MIVCs are explored since the size of `Unexplored` is reduced after every iteration. What remains to be shown is that every output set is an MIVC and that every such MIVC is fresh, i.e., produced only once. Since GROW-SHRINK outputs results only during shrinking, we examine the shrinking procedure. Recall that the input condition of `Shrink`(N) is that N is adequate and all proper subsets of N are unexplored (Section 9.3.1), and the output condition is that result N_{mivc} is an MIVC, and $N_{mivc} = N$ or N_{mivc} is unexplored.

We shrink two kinds of adequate sets in our algorithm: either a set U_{ivc} that is a subset of an adequate maximal unexplored subset U_{max} (procedure findMIVCs, line 6), or a set U that was stored in *shrinkingQueue* (procedure findMIVCs, line 11). In the former case, by Proposition 2.10 every adequate subset of U_{max} (and hence also every adequate subset of U_{ivc}) is unexplored, thus we satisfy the input condition of the shrinking procedure, and by the output condition the resultant set is an unexplored MIVC (i.e., a fresh MIVC).

However, in the latter case, all the sets stored in *shrinkingQueue* are already explored. Here, we exploit the invariants I1) and I2) about *shrinkingQueue*. Assume that we satisfy the input condition of shrinking. Therefore, the resultant MIVC is unexplored (and thus fresh) or it equals to the input set U, and by I1), there is no already produced MIVC X such that $X \subseteq U$, hence U is fresh. Now, we show that we satisfy the input condition, i.e., that all proper adequate subsets of U are unexplored. By contradiction, assume that there is an adequate proper explored subset U' of U. Observe that in GROW-SHRINK we mark adequate sets as explored only in two situations: either U' is a superset of a produced MIVC (procedure shrink, line 14) or U' is a superset of a set Y that was added to *shrinkingQueue* (procedure grow, line 7). However, since $U' \subsetneq U$, due to I1), there is not such produced MIVC, and due to I2), there is no such Y in *shrinkingQueue*.

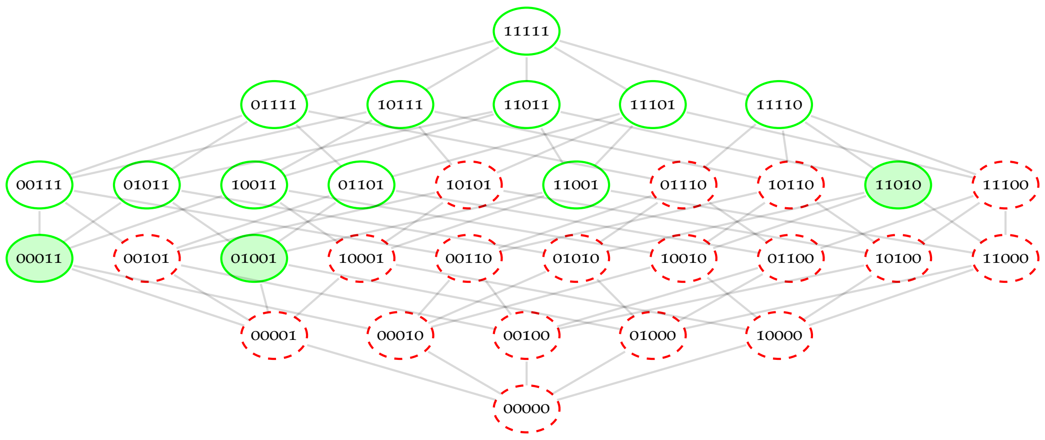


Figure 9.1: The power-set from the example execution of our algorithm.

9.4 *Implementation*

We have implemented the GROW-SHRINK algorithm in an industrial model checker called `JKind` [Gacek et al., 2018], which verifies safety properties of infinite-state synchronous systems. It accepts Lustre programs [Halbwachs et al., 1991] as input. The translation of Lustre into a symbolic transition system in `JKind` is straightforward and is similar to what is described in [Hagen and Tinelli, 2008]. Verification is supported by multiple "proof engines" that execute in parallel, including K-induction, property directed reachability (PDR), and lemma generation. During verification, `JKind` emits SMT problems using the theories of linear integer and real arithmetic, and can use the `Z3` [de Moura and Bjørner, 2008], `Yices` [Dutertre and Moura, 2006], `MathSAT` [Cimatti et al., 2013], `SMTInterpol` [Christ et al., 2012], and `CVC4` [Barrett et al., 2011] SMT solvers as back-ends. When a property is proved and IVC generation is enabled, an additional parallel engine executes the IVC_UC algorithm [Ghassabani et al., 2016] to generate an (approximately) minimal IVC. To implement our method, we have extended `JKind` with a new engine that implements Algorithm 9.5 on top of `Z3` [de Moura and Bjørner, 2008]. We use the `JKind` IVC generation engine to implement the procedure IVC_UC in Algorithm 9.5. The source code is publicly available at:

`https://github.com/jar-ben/online-mivc-enumeration`

9.5 *Example Execution*

The following example explains the execution of our algorithm on a simple instance where the transition step predicate T is given as a conjunction of five sub-predicates $\{T_1, T_2, T_3, T_4, T_5\}$. We do not exactly state what are the predicates and what is the safety property of interest. Instead, Figure 9.1 illustrates the power set of $\{T_1, T_2, T_3, T_4, T_5\}$ together with an information about adequacy of individual subsets. The subsets with solid green border are the adequate subsets, and the subsets with dashed red border are the inadequate ones. To save space, we encode subsets as bitvectors, for example the subset

$\{T_1, T_2, T_4\}$ is written as 11010. There are three MIVCs in this example: 00011, 01001, and 11010.

We illustrate the first iteration of the main procedure findMIVCs of our algorithm. Initially, all subsets are unexplored, i.e. $map^+ \wedge map^- = True$ and the queue *shrinkingQueue* is empty. The procedure starts by finding a maximal unexplored subset and checking it for adequacy. In our case, $U_{max} = 11111$ is the only maximal unexplored subset and it is determined to be adequate. Thus, the algorithm IVC_UC is used to compute an approximately minimal IVC $U_{IVC} = 01101$ which is then shrunk to an MIVC 01001.

During the shrinking, sets 00101, 01001, and 01000 are subsequently checked for adequacy and determined to be inadequate, adequate, and inadequate, respectively. The set 01001 is the resultant MIVC, thus the formula $map^+ \wedge map^-$ is updated to $map^+ \wedge map^- = True \wedge (\neg x_2 \vee \neg x_5)$. The other two sets, 00101 and 01000, are enqueued to the *growingQueue* and grown at the end of the procedure.

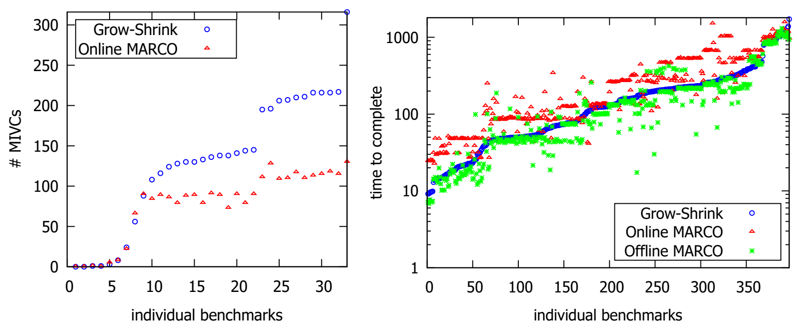
We first grow the set 00101. Initially, the procedure grow picks $M = 10111$ as the maximal unexplored superset of 00101, and checks it for adequacy. It is adequate and thus, an approximately minimal IVC $M_{IVC} = 00011$ is computed, enqueued to *shrinkingQueue*, and formula $map^+ \wedge map^-$ is updated to $map^+ \wedge map^- = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5)$. Then, M is (based on M_{IVC}) reduced to $M = 10101$ and checked for adequacy. It is found to be inadequate, thus formula $map^+ \wedge map^-$ is updated to $map^+ \wedge map^- = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4)$, and the procedure terminates.

The growing of the set 01000 results into an approximately maximal inadequate subset 01110. Moreover, an approximately minimal IVC 11110 is found during the growing and enqueued into *shrinkingQueue*. The formula $map^+ \wedge map^-$ is updated to $map^+ \wedge map^- = True \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee x_5)$.

After the second grow, the procedure shrink terminates and the main procedure findMIVCs continues. The queue *shrinkingQueue* contains two sets: 00011, 11110, thus the procedure now shrinks them. During shrinking the set 00011, the algorithm would attempt to check the sets 00001 and 00010 for adequacy, however since both these are already explored, the set 00011 is identified to be an MIVC without performing any adequacy checks. The procedure findMIVCs would now shrink also the last set 11110 in the queue *shrinkingQueue*, and continue with a next iteration.

9.6 Experimental Evaluation

We now experimentally compare GROW-SHRINK, the algorithm presented in this chapter, and the two state-of-the-art algorithms (briefly described in Section 9.2): OfflineMARCO, the algorithm from [Ghassabani et al., 2017b], and OnlineMARCO, a variant of the algorithm from [Ghassabani et al., 2017b] that performs a shrink step prior to



(a) (b)

Figure 9.2: Figure (a) shows the number of produced MIVCs by online algorithms. Figure (b) shows the runtime for tractable benchmarks in a log scale.

returning a solution to ensure minimality. We investigate the following research questions:

RQ1: For the large models where the complete MIVC enumeration is intractable, how many MIVCs are found within the given time limit?

RQ2: For the tractable models, i.e. models in which all MIVCs are found, how much time is required to complete the enumeration of MIVCs?

RQ3: What is the (average) number of solver (model-checking) calls with result adequate/inadequate required by evaluated online algorithms to produce individual MIVCs?

Experimental Setup We start from a benchmark suite that is a superset of the benchmarks used in [Ghassabani et al., 2017b]. This suite contains 660 models, and includes all models that yield a valid result (530 in total) from previous Lustre model checking papers [Hagen and Tinelli, 2008, Mebsout and Tinelli, 2016] and 130 industrial models yielding valid results derived from an infusion pump system [Murugesan et al., 2013] and other sources [Mebsout and Tinelli, 2016, Backes et al., 2015]. As this chapter is concerned with analyzing problems involving multiple MIVCs, we include only models that had more than 4 MIVCs (46 models in total). To consider problems with many IVCs, we took those models and mutated them, constructing 20 mutants for each model. The mutants varied both in the number and in the size of individual MIVCs. We added the mutants that still yielded valid results and have more than 5 MIVCs (384 in total) back to the benchmark suite. Thus, the final suite contains 430 Lustre models. The original benchmarks and our augmented benchmark are available online[5].

For each test model, we configured `JKind` to use the `Z3` solver and the "fastest" mode of `JKind` (which involves running the k-induction and PDR engines in parallel and terminating when a solution is found). The experiments were run on a 3.50GHz Intel(R) i5-4690 processor 16 GB memory machine running Linux with a 30 minute timeout. All experimental data is available online[6].

[5] `https://github.com/elaghs/benchmarks`

[6] `https://github.com/jar-ben/online-mivc-enumeration`

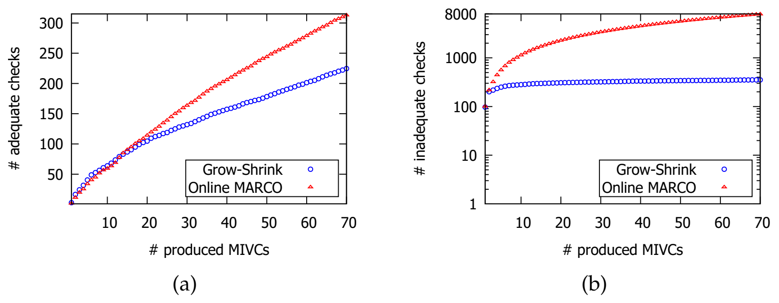


Figure 9.3: Average number of performed adequacy checks required to produce individual MIVCs. Figure (a) shows the number of checks with result "adequate", and Figure (b) the number of checks with result "inadequate". Note that Figure (b) is in a log scale.

9.6.1 *Experimental Results*

RQ1 and RQ2 Data related to the first two research questions are shown in Figures 9.2(a) and 9.2(b). Figure 9.2(a) describes the number of MIVCs found be the two online algorithms in the intractable benchmarks, i.e. the benchmarks where the algorithms did not complete the computation within the time limit. There are 33 such benchmarks. GROW-SHRINK substantially outperforms OnlineMARCO in the majority of the benchmarks, finding an average of 55% additional MIVCs.

In Figure 9.2(b), we show the time for each algorithm needed to complete the MIVC enumeration in case of the 397 tractable benchmarks. GROW-SHRINK is on average only 1.08 times slower than OfflineMARCO, yet as previously discussed, it has the advantage of returning guaranteed MIVCs, rather than approximate MIVCs, already during the computation. Compared to OnlineMARCO, GROW-SHRINK is on average 1.5 times faster.

RQ3 For RQ3, we examined the number of required calls to the solver per MIVC. For this question, we used the 33 models that contained a large number of MIVCs (>70) in order to show the solver efficiency as the number of MIVCs increased. A point with coordinates (x, y) states that the algorithm needed to perform y solver calls (on average) in order to produce (find) the first x MIVCs. We grouped the calls in terms of the number of calls that returned *adequate* vs. *inadequate* results. It is evidenced by the results in Figure 9.3, the new algorithm improves upon OnlineMARCO as the number of MIVCs becomes larger.

The improvement in the number of *inadequate* calls is due the novel shrinking and growing procedures. Each (approximately) maximal inadequate subset found by the growing procedure allows to save (up to exponentially) many inadequate calls during subsequent executions of the shrinking procedure. Indeed, GROW-SHRINK performed on average only 353 inadequate calls to output the first 70 MIVCs, whereas OnlineMARCO needed to perform 7775 calls to output the same number of MIVCs.

The improvement in the number of *adequate* calls is not so significant as in the case of inadequate calls. Yet, since the adequate calls are usually much more time consuming than inadequate ones, even a slight saving in the number of adequate calls might significantly speed up the whole computation. GROW-SHRINK saves adequate calls due to the usage of the shrinking queue and due to the invariants that are maintained by the queue. In particular, shall two comparable sets appear in the queue, only the smaller is left. Thus, the algorithm avoids shrinking of relatively large sets and saves some adequate calls.

9.7 *Summary and Future Work*

We have presented an *online* algorithm, called GROW-SHRINK, for computation of minimal Inductive Validity Cores (MIVCs). The new algorithm substantially outperforms previous approaches. As opposed to the OfflineMARCO algorithm in [Ghassabani et al., 2017b], it is guaranteed to produce minimal IVCs. As opposed to a naive extension OnlineMARCO, the new algorithm is substantially faster and requires fewer solver calls as the number of MIVCs increases. We believe that this new algorithm will substantially increase the applicability of software engineering tasks that require MIVCs.

In the future, we hope to examine parallel computation of MIVCs using a variant of this algorithm to further increase scalability. Another possible direction is to evaluate GROW-SHRINK for other instances of MSMPs (minimal sets over a monotone predicate). In particular, note that besides the approximate MIVC extractor IVC_UC, GROW-SHRINK does not directly exploit any specific properties of the MIVC domain. Hence, GROW-SHRINK can be straightforwardly used for other instances of MSMPs. So far, we have tried to use it only for the task of finding minimal unsatisfiable subsets in the Boolean CNF domain (Chapter 5), and GROW-SHRINK did not perform very well compared to enumerators that are tailored to MUSes. However, based on our evaluation of domain agnostic MUS/MSMP enumeration algorithms (Section 4.7), the performance of individual algorithms vary a lot for various instances of MSMPs. Hence, it is very likely that GROW-SHRINK would be efficient for some instances of MSMPs.

Part V

Minimal Sufficient Reductions

10 Relaxing Timed Automata For Reachability

In the last chapter of the thesis, we study a yet another instance of minimal sets over a monotone predicate called *minimal sufficient reductions (MSRs)*. MSR is our novel concept [Bendík et al., 2021] that find an application in the area of timed automata.

A timed automaton (TA) [Alur and Dill, 1994] is a finite automaton extended with a set of real-time variables, called clocks, which capture the time. The clocks enrich the semantics and the constraints on the clocks restrict the behavior of the automaton, which are particularly important in modeling time-critical systems. The examples of TA models of critical systems include scheduling of real-time systems [Fehnker, 1999, David et al., 2009, Guan et al., 2007], medical devices [Kwiatkowska et al., 2015, Jiang et al., 2014], and rail-road crossing systems [Wang, 2004].

Model-checking methods allow for verifying whether a given TA meets a given system specification. Contemporary model-checking tools, such as UPPAAL [Behrmann et al., 2006] or Imitator [André et al., 2012], have proved to be practically applicable on various industrial case studies [Behrmann et al., 2006, André et al., 2019b, Henzinger et al., 2001]. Unfortunately, during the system design phase, the system information is often incomplete. A designer is often able to build a TA with correct structure, i.e., exactly capturing locations and transitions of the modeled system, however the exact clock (timing) constraints that enable/trigger the transitions are uncertain. Thus, the produced TA often does not meet the specification (i.e., it does not pass the model-checking) and it needs to be fixed. If the specification declares universal properties, e.g., safety or unavoidability, that need to hold on each trace of the TA, a model-checker either returns "yes", or it returns "no" and generates a trace, called "counter example", along which the property is violated. This trace can be used to repair the model in an automated way [Kölbl et al., 2019]. However, in the case of existential properties, such as reachability, the property has to hold on a trace of the TA. The model-checker either returns "yes" and generates a witness trace satisfying the property, or returns just "no" and does not provide any additional information that would help the designer to correct the TA.

Contribution. In this chapter, we study the following problem: given a timed automaton $\mathcal{A}$ and a reachability property that is not satisfied by $\mathcal{A}$, relax clock constraints of $\mathcal{A}$ such that the resultant automaton $\mathcal{A}'$ satisfies the reachability property. Moreover, the goal is to minimize the number of the relaxed clock constraints and, secondary, also to minimize the overall change of the timing constants used in the clock constraints. We propose a two step solution for this problem. In the first step, we identify a *minimal sufficient reduction (MSR)* of $\mathcal{A}$, i.e., an automaton $\mathcal{A}''$ that satisfies the reachability property and originates from $\mathcal{A}$ by removing only a minimal necessary set of clock constraints. In the second step, instead of completely removing the clock constraints, we employ mixed integer linear programming (MILP) to find a minimal relaxation of the constraints that leads to a satisfaction of the reachability property along a witness path.

The underlying assumption is that during the design phase the most suitable timing constants reflecting the system properties are defined. Thus, our goal is to generate a TA satisfying the reachability property by changing a minimum number of timing constants. Some of the constraints of the initial TA can be strict (no relaxation is possible), which can easily be integrated to the proposed solution. Thus, the proposed method can be viewed as a way to handle design uncertainties: develop a TA $\mathcal{A}$ in a best-effort basis and apply our algorithm to find a $\mathcal{A}'$ that is *as close as* possible to $\mathcal{A}$ and satisfies the given reachability property.

Related Work. Another way to handle uncertainties about timing constants is to build a *parametric* timed automata (PTA), i.e., a TA where clock constants can be represented with parameters. Subsequently, a parameter synthesis tool, such as [Lime et al., 2009, André et al., 2012, Bezděk et al., 2018], can be used to find suitable values of the parameters for which the resultant TA satisfies the specification. However, most of the parameter synthesis problems are undecidable [André, 2019]. While symbolic algorithms without termination guarantees exist for some subclasses [Bezděk et al., 2016, Jovanovic et al., 2015, André et al., 2019c], these algorithms are computationally very expensive compared to model checking (see [André, 2018]). Moreover, minimizing the number of modified clock constraints is not straightforward.

A related TA repair problem has been studied in a recent work [André et al., 2019a], where the authors also assumed that some of the constraints are incorrect. To repair the TA, they parametrized the initial TA and generated parameters by analyzing traces of the TA. However, the authors [André et al., 2019a] do not focus on repairing the TA w.r.t. reachability properties as we do. Instead, their goal is to make the TA compliant with an oracle that decides if a trace of the TA belongs to a system or not. Thus, their approach can not handle reachability properties. Furthermore in [André et al., 2019a], the total change of the timing constraints is minimized, while we primarily minimize the number of changed constraints, then the total change.

10.1 *Preliminaries*

10.1.1 *Timed Automata*

A *timed automaton* (TA) [Alur and Dill, 1994, Larsen and Yi, 1997, Alur, 1999] is a finite-state machine extended with a finite set C of real-valued clocks. A *clock* $x \in C$ measures the time spent after its last reset. In a TA, clock constraints are defined for locations (states) and transitions. A *simple clock constraint* is defined as $x - y \sim c$ where $x, y \in C \cup \{0\}$, $\sim \in \{<, \leqslant\}$ and $c \in \mathbb{Z} \cup \{\infty\}$.[1] Simple clock constraints and constraints obtained by combining these with conjunction operator ($\wedge$) are called *clock constraints*. The sets of simple and all clock constrains are denoted by $\Phi_S(C)$ and $\Phi(C)$, respectively. For a clock constraint $\phi \in \Phi(C)$, $\mathcal{S}(\phi)$ denotes the simple constraints from ϕ, e.g., $\mathcal{S}(x - y < 10 \wedge y \leqslant 20) = \{x - y < 10, y \leqslant 20\}$. A *clock valuation* $v : C \to \mathbb{R}_+$ assigns non-negative real values to each clock. The notation $v \models \phi$ denotes that the clock constraint ϕ evaluates to true when each clock x is replaced with $v(x)$. For a clock valuation v and $d \in \mathbb{R}_+$, $v + d$ is the clock valuation obtained by *delaying* each clock by d, i.e., $(v + d)(x) = v(x) + d$ for each $x \in C$. For $\lambda \subseteq C$, $v[\lambda := 0]$ is the clock valuation obtained after *resetting* each clock from λ, i.e., $v[\lambda := 0](x) = 0$ for each $x \in \lambda$ and $v[\lambda := 0](x) = v(x)$ for each $x \in C \backslash \lambda$.

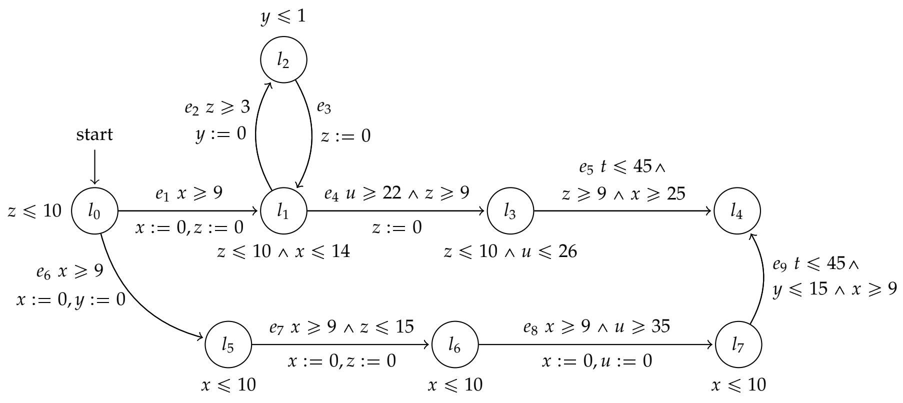
[1] Simple constraints are only defined as upper bounds to simplify the presentation. This definition is not restrictive since $x - y \geqslant c$ and $x \geqslant c$ are equivalent to $y - x \leqslant -c$ and $0 - x \leqslant -c$, respectively. A similar argument holds for strict inequality ($>$).

Definition 10.1 (timed automata). *A timed automaton (TA) is a tuple $\mathcal{A} = (L, l_0, C, \Delta, Inv)$, where L is a finite set of locations, $l_0 \in L$ is the initial location, C is a finite set of clocks, $\Delta \subseteq L \times \mathcal{P}(C) \times \Phi(C) \times L$ is a finite transition relation, and $Inv : L \to \Phi(C)$ is an invariant function.*

For a transition $e = (l_s, \lambda, \phi, l_t) \in \Delta$, l_s is the source location, l_t is the target location, λ is the set of clocks reset on e and ϕ is the guard (i.e., a clock constraint) tested for enabling e. The semantics of a TA is given by a labelled transition system (LTS). An LTS is a tuple $\mathcal{T} = (S, s_0, \Sigma, \to)$, where S is a set of states, $s_0 \in S$ is an initial state, Σ is a set of symbols, and $\to \subseteq S \times \Sigma \times S$ is a transition relation. A transition $(s, a, s') \in \to$ is also shown as $s \xrightarrow{a} s'$.

Definition 10.2 (LTS semantics for TA). *Given a timed automaton $\mathcal{A} = (L, l_0, C, \Delta, Inv)$, the labelled transition system $T(\mathcal{A}) = (S, s_0, \Sigma, \to)$ is defined as follows:*

- $S = \{(l, v) \mid l \in L, v \in \mathbb{R}_+^{|C|}, v \models Inv(l)\}$,
- $s_0 = (l_0, \boldsymbol{o})$, *where* $\boldsymbol{o}(x) = 0$ *for each* $x \in C$,
- $\Sigma = \{act\} \cup \mathbb{R}_+$, *and*
- *the transition relation $\to$ is defined by the following rules:*
  - *delay transition:* $(l, v) \xrightarrow{d} (l, v + d)$ *if* $v + d \models Inv(l)$
  - *discrete transition:* $(l, v) \xrightarrow{act} (l', v')$ *if there exists* $(l, \lambda, \phi, l') \in \Delta$ *such that* $v \models \phi$, $v' = v[\lambda := 0]$, *and* $v' \models Inv(l')$.



$y \leqslant 1$
l_2
$e_2\ z \geqslant 3$
$y := 0$
e_3
$z := 0$
start
$z \leqslant 10$
l_0
$e_1\ x \geqslant 9$
$x := 0, z := 0$
l_1
$z \leqslant 10 \wedge x \leqslant 14$
$e_4\ u \geqslant 22 \wedge z \geqslant 9$
$z := 0$
l_3
$z \leqslant 10 \wedge u \leqslant 26$
$e_5\ t \leqslant 45 \wedge$
$z \geqslant 9 \wedge x \geqslant 25$
l_4
$e_9\ t \leqslant 45 \wedge$
$y \leqslant 15 \wedge x \geqslant 9$
$e_6\ x \geqslant 9$
$x := 0, y := 0$
l_5
$x \leqslant 10$
$e_7\ x \geqslant 9 \wedge z \leqslant 15$
$x := 0, z := 0$
l_6
$x \leqslant 10$
$e_8\ x \geqslant 9 \wedge u \geqslant 35$
$x := 0, u := 0$
l_7
$x \leqslant 10$

Figure 10.1: An example of a timed automaton.

The notation $s \rightarrow_d s'$ is used to denote a delay transition of duration d followed by a discrete transition from s to s', i.e., $s \xrightarrow{d} s \xrightarrow{act} s'$. A run ρ of $\mathcal{A}$ is either a finite or an infinite alternating sequence of delay and discrete transitions, i.e., $\rho = s_0 \rightarrow_{d_0} s_1 \rightarrow_{d_1} s_2 \rightarrow_{d_2} \cdots$. The set of all runs of $\mathcal{A}$ is denoted by $[[\mathcal{A}]]$.

A path π of $\mathcal{A}$ is an interleaving sequence of locations and transitions, $\pi = l_0, e_1, l_1, e_2, \ldots$, where $e_{i+1} = (l_i, \lambda_{i+1}, \phi_{i+1}, l_{i+1}) \in \Delta$ for each $i \geqslant 0$. Furthermore, a path $\pi = l_0, e_1, l_1, e_2, \ldots$ of $\mathcal{A}$ is said to be *realizable* if there exists a delay sequence $d_0, d_1, \ldots$ such that $(l_0, \mathbf{0}) \rightarrow_{d_0} (l_1, v_1) \rightarrow_{d_1} (l_1, v_2) \rightarrow_{d_2} \cdots$ is a run of $\mathcal{A}$ and for every $i \geqslant 1$, the ith discrete transition is taken according to e_i, i.e., $e_i = (l_{i-1}, \lambda_i, \phi_i, l_i)$, $v_{i-1} + d_{i-1} \models \phi_i$, $v_i = (v_{i-1} + d_{i-1})[\lambda_i := 0]$ and $v_i \models Inv'(l_i)$.

For a TA $\mathcal{A}$ and a subset of its locations $L_T \subsetneq L$, L_T is *reachable* on $\mathcal{A}$ if there exists $\rho = (l_0, \mathbf{0}) \rightarrow_{d_0} (l_1, v_1) \rightarrow_{d_1} \ldots \rightarrow_{d_{n-1}} (l_n, v_n) \in [[\mathcal{A}]]$ such that $l_n \in L_T$; otherwise, L_T is *unreachable*. The reachability problem, *isReachable*$(\mathcal{A}, L_T)$, is decidable and implemented in various verification tools including UPPAAL [Behrmann et al., 2006]. The verifier either returns “No” indicating that such a run does not exist, or it generates a run (witness) satisfying the reachability requirement.

Example 10.1. *Figure 10.1 illustrates a TA with 8 locations: $\{l_0, \ldots, l_7\}$, 9 transitions: $\{e_1, \ldots, e_9\}$, an initial location l_0, and an unreachable set of locations $L_T = \{l_4\}$.*

10.1.2 *Timed Automata Relaxations and Reductions*

For a timed automaton $\mathcal{A} = (L, l_0, C, \Delta, Inv)$, the set of pairs of transition and associated simple constraints are defined as follows:

$$\Psi(\Delta) = \{(e, \varphi) \mid e = (l_s, \lambda, \phi, l_t) \in \Delta, \varphi \in \mathcal{S}(\phi)\} \tag{10.1}$$

The set of pairs of location and associated simple constraints is defined as:

$$\Psi(Inv) = \{(l, \varphi) \mid l \in L, \varphi \in \mathcal{S}(Inv(l))\} \tag{10.2}$$

Definition 10.3 (constraint-relaxation). *Let $\phi \in \Phi(C)$ be a constraint over C, $\Theta \subseteq \mathcal{S}(\phi)$ be subset of its simple constraints and $\mathbf{r} : \Theta \to \mathbb{N} \cup \{\infty\}$ be a positive valued relaxation valuation. The relaxed constraint is defined as:*

$$R(\phi, \Theta, \mathbf{r}) = \left(\bigwedge_{\varphi \in \mathcal{S}(\phi) \setminus \Theta} \varphi \right) \wedge \left(\bigwedge_{\varphi = x - y \sim c \in \Theta} x - y \sim c + \mathbf{r}(\varphi) \right) \quad (10.3)$$

Intuitively, $R(\phi, \Theta, \mathbf{r})$ relaxes only the thresholds of simple constraints from Θ with respect to $\mathbf{r}$, e.g., $R(x - y \leqslant 10 \wedge y < 20, \{y < 20\}, \mathbf{r}) = x - y \leqslant 10 \wedge y < 23$, where $\mathbf{r}(y < 20) = 3$. Setting a threshold to ∞ implies removing the corresponding simple constraint, e.g., $R(x - y \leqslant 10 \wedge y < 20, \{y < 20\}, \mathbf{r}) = x - y \leqslant 10$, where $\mathbf{r}(y < 20) = \infty$. Note that $R(\phi, \Theta, \mathbf{r}) = \phi$ when Θ is empty.

Definition 10.4 ($(D, I, \mathbf{r})$-relaxation). *Let $\mathcal{A} = (L, l_0, C, \Delta, Inv)$ be a timed automaton, $D \subseteq \Psi(\Delta)$ and $I \subseteq \Psi(Inv)$ be transition and location constraint sets, and $\mathbf{r} : D \cup I \to \mathbb{N} \cup \{\infty\}$ be a positive valued relaxation valuation. The $(D, I, \mathbf{r})$-relaxation of $\mathcal{A}$, denoted $\mathcal{A}_{<D,I,\mathbf{r}>}$, is a TA $\mathcal{A}' = (L', l'_0, C', \Delta', Inv')$ such that:*

- *$L = L'$, $l_0 = l'_0$, $C = C'$, and*
- *Δ' originates from Δ by relaxing D via $\mathbf{r}$. For $e = (l_s, \lambda, \phi, l_t) \in \Delta$, let $D|_e = \{\varphi \mid (e, \varphi) \in D\}$, $\mathbf{r}|_e(\varphi) = \mathbf{r}(e, \varphi)$, then*

$$\Delta' = \{(l_s, \lambda, R(\phi, D|_e, \mathbf{r}|_e), l_t) \mid e = (l_s, \lambda, \phi, l_t) \in \Delta\}$$

- *Inv' originates from Inv by relaxing I via $\mathbf{r}$. For $l \in L$, let $I|_l = \{\varphi \mid (l, \varphi) \in I\}$, and $\mathbf{r}|_l(\varphi) = \mathbf{r}(l, \varphi)$, then $Inv'(l) = R(Inv(l), I|_l, \mathbf{r}|_l)$.*

Intuitively, the TA $\mathcal{A}_{<D,I,\mathbf{r}>}$ emerges from a TA $\mathcal{A}$ by relaxing the guards of the transitions from the set D and relaxing invariants of the locations from I with respect to $\mathbf{r}$. In the special case of setting the threshold of each constraint from D and I to ∞, i.e., when $\mathbf{r}(a) = \infty$ for each $a \in D \cup I$, the corresponding simple constraints are effectively removed, which is called a (D,I)-reduction and denoted by $\mathcal{A}_{<D,I>}$. Note that $\mathcal{A} = \mathcal{A}_{<\emptyset,\emptyset>}$.

Proposition 10.1. *Let $\mathcal{A} = (L, l_0, C, \Delta, Inv)$ be a timed automaton, $D \subseteq \Psi(\Delta)$ and $I \subseteq \Psi(Inv)$ be sets of simple guard and invariant constraints, and $\mathbf{r} : D \cup I \to \mathbb{N} \cup \{\infty\}$ be a relaxation valuation. Then $[[\mathcal{A}]] \subseteq [[\mathcal{A}_{<D,I,\mathbf{r}>}]]$.*

Proof. First, observe that for a clock constraint $\phi \in \Phi(C)$, a subset of its simple constraints $\Theta \subseteq \mathcal{S}(\phi)$, a relaxation valuation $\mathbf{r}'$ for Θ, and the relaxed constraint $R(\phi, \Theta, \mathbf{r}')$ as in Definition 10.3, it holds that

$$\text{for any clock valuation } v : \quad v \models \phi \Rightarrow v \models R(\phi, \Theta, \mathbf{r}'). \quad (10.4)$$

Now, consider a run $\rho = (l_0, \mathbf{0}) \to_{d_0} (l_1, v_1) \to_{d_1} (l_2, v_2) \to_{d_2} \cdots \in [[\mathcal{A}]]$. Let $\pi = l_0, e_1, l_1, e_2, \ldots$ with $e_i = (l_{i-1}, \lambda_i, \phi_i, l_i) \in \Delta$ for each $i \geqslant 1$ be the path realized as ρ via delay sequence $d_0, d_1, \ldots$. By Definition 10.4

for each $(l, \lambda, \phi, l') \in \Delta$, there is $(l, \lambda, R(\phi, D|_e, \mathbf{r}|_e), l') \in \Delta'$. We define a path induced by π on $\mathcal{A}_{<D,I,\mathbf{r}>}$ as:

$$M(\pi) = l_0, (l_0, \lambda_1, R(\phi_1, D|_{e_1}, \mathbf{r}|_{e_1}), l_1), l_1, (l_1, \lambda_2, R(\phi_2, D|_{e_2}, \mathbf{r}|_{e_2}), l_2, \dots \tag{10.5}$$

By (10.4), for each $i = 0, \dots, n-1$ it holds that $v_i \models R(Inv(l_i), D|_{l_i}, \mathbf{r}|_{l_i})$, $v_i + d_i \models R(Inv(l_i), D|_{l_i}, \mathbf{r}|_{l_i})$ and $v_i + d_i \models R(\phi_{i+1}, D|_{e_{i+1}}, \mathbf{r}|_{e_{i+1}})$. Thus $M(\pi)$ is realizable on $\mathcal{A}_{<D,I,\mathbf{r}>}$ via the same delay sequence and $\rho \in [[\mathcal{A}_{<D,I,\mathbf{r}>}]]$. As $\rho \in [[\mathcal{A}]]$ is arbitrary, we conclude that $[[\mathcal{A}]] \subseteq [[\mathcal{A}_{<D,I,\mathbf{r}>}]]$. □

10.1.3 *Problem Statement*

Problem 10.1. *Given a TA $\mathcal{A} = (L, l_0, C, \Delta, Inv)$ and a set of target locations $L_T \subsetneq L$ that is unreachable on $\mathcal{A}$, find a $(D, I, \mathbf{r})$-relaxation $\mathcal{A}_{<D,I,\mathbf{r}>}$ of $\mathcal{A}$ such that L_T is reachable on $\mathcal{A}_{<D,I,\mathbf{r}>}$. Moreover, the goal is to minimize the number $|D \cup I|$ of constraints that are relaxed, and, secondary, we tend to minimize the overall change of the clock constraints $\sum_{c \in D \cup I} \mathbf{r}(c)$.*

We propose a two step solution to this problem. In the first step, we identify a minimal subset $D \cup I$ of the simple constraints $\Psi(\Delta) \cup \Psi(Inv)$ such that L_T is reachable on the (D, I)-reduction $\mathcal{A}_{<D,I>}$. Consequently, we can obtain a witness path of the reachability on $\mathcal{A}_{<D,I>}$ from the verifier. The path would be realizable on $\mathcal{A}$ if we remove the constraints $D \cup I$. In the second step, instead of completely removing the constraints $D \cup I$, we find a relaxation valuation $\mathbf{r} : D \cup I \to \mathbb{N} \cup \{\infty\}$ such that the path found in the first step is realizable on $\mathcal{A}_{<D,I,\mathbf{r}>}$. To find $\mathbf{r}$, we introduce relaxation parameters for constraints in $D \cup I$. Subsequently, we solve an MILP problem to find a valuation of the parameters, i.e., $\mathbf{r}$, that makes the path realizable on $\mathcal{A}_{<D,I,\mathbf{r}>}$ and minimizes $\sum_{c \in D \cup I} \mathbf{r}(c)$. Note that it might be the case that the reduction $\mathcal{A}_{<D,I>}$ contains multiple realizable paths that lead to L_T, and another path might result in a smaller overall change. While our approach can be applied to a number of paths, processing all of them can be practically intractable.

10.2 *Minimal Sufficient (D,I)-Reductions*

Throughout this section, let us simply write a *reduction* when talking about a (D,I)-reduction of $\mathcal{A}$. We use two notations for naming a reduction; either we simply use capital letters, e.g., M, N, K to name a reduction, or we use the notation $\mathcal{A}_{<D,I>}$ to also specify the sets D, I of simple clock constraints. Given a reduction $N = \mathcal{A}_{<D,I>}$, we write $|N|$ to denote the cardinality $|D \cup I|$. Furthermore, let us by $\mathcal{R}_{\mathcal{A}}$ denote the set of all reductions. We define a partial order relation $\sqsubseteq$ on $\mathcal{R}_{\mathcal{A}}$ as $\mathcal{A}_{<D,I>} \sqsubseteq \mathcal{A}_{<D',I'>}$ iff $D \cup I \subseteq D' \cup I'$. Similarly, we write $\mathcal{A}_{<D,I>} \sqsubsetneq \mathcal{A}_{<D',I'>}$ iff $D \cup I \subsetneq D' \cup I'$. We say that a reduction $\mathcal{A}_{<D,I>}$ is a *sufficient reduction* (w.r.t. $\mathcal{A}$ and L_T) iff L_T is reachable on $\mathcal{A}_{<D,I>}$; otherwise, $\mathcal{A}_{<D,I>}$ is an *insufficient reduction*. Crucial observation for our work is that the property of being a sufficient reduction is monotone w.r.t. the partial order:

Proposition 10.2. *Let $\mathcal{A}_{<D,I>}$ and $\mathcal{A}_{<D',I'>}$ be reductions of $\mathcal{A}$ such that $\mathcal{A}_{<D,I>} \sqsubseteq \mathcal{A}_{<D',I'>}$. If $\mathcal{A}_{<D,I>}$ is sufficient then $\mathcal{A}_{<D',I'>}$ is also sufficient.*

Proof. Note that $\mathcal{A}_{<D',I'>}$ is a $(D' \backslash D, I' \backslash I)$-reduction of $\mathcal{A}_{<D,I>}$. By Proposition 10.1, $[[\mathcal{A}_{<D,I>}]] \subseteq [[\mathcal{A}_{<D',I'>}]]$, i.e., the run of $\mathcal{A}_{<D,I>}$ that witnesses the reachability of L_T is also a run of $\mathcal{A}_{<D',I'>}$. □

Definition 10.5 (MSR). *A sufficient reduction $\mathcal{A}_{<D,I>}$ is a* minimal sufficient reduction (MSR) *iff there is no $c \in D \cup I$ such that the reduction $\mathcal{A}_{<D \backslash \{c\}, I \backslash \{c\}>}$ is sufficient. Equivalently, due to Proposition 10.2, $\mathcal{A}_{<D,I>}$ is an MSR iff there is no sufficient reduction $\mathcal{A}_{<D',I'>}$ such that $\mathcal{A}_{<D',I'>} \sqsubsetneq \mathcal{A}_{<D,I>}$.*

Recall that a reduction $\mathcal{A}_{<D,I>}$ is determined by $D \subseteq \Psi(\Delta)$ and $I \subseteq \Psi(Inv)$. Consequently, $|\mathcal{R}_{\mathcal{A}}| = 2^{|\Psi(\Delta) \cup \Psi(Inv)|}$. Moreover, there can be up to $\binom{k}{k/2}$ MSRs where $k = |\Psi(\Delta) \cup \Psi(Inv)|$ (see Sperner's theorem [Sperner, 1928]). Also note, that the *minimality* of a reduction does not mean a *minimum* number of simple clock constraints that are reduced by the reduction; there can exist two MSRs, M and N, such that $|M| < |N|$. Since our overall goal is to relax $\mathcal{A}$ as least as possible, we identify a *minimum* MSR, i.e., an MSR M such that there is no MSR M' with $|M'| < |M|$, and then use the minimum MSR for the MILP part (Section 10.3) of our overall approach. There can be also up to $\binom{k}{k/2}$ minimum MSRs.

Example 10.2. *Assume the TA $\mathcal{A}$ from Example 10.1 (illustrated in Figure 10.1) and let $L_T = \{l_4\}$ which is unreachable on $\mathcal{A}$. There are 24 MSRs and 4 of them are minimum. For example, $\mathcal{A}_{<D,I>}$ with $D = \{(e_5, x \geqslant 25)\}$ and $I = \{(l_3, u \leqslant 26)\}$ is a minimum MSR, and $\mathcal{A}_{<D',I'>}$ with $D' = \{(e_9, y \leqslant 15), (e_7, z \leqslant 15)\}$ and $I' = \{(l_6, x \leqslant 10)\}$ is a non-minimum MSR.*

Observe, that MSRs are an instance of minimal sets over a monotone predicate:

Observation 10.1. *Minimal sufficient reductions (MSRs) are an instance of Minimal Sets over a Monotone Predicate (MSMPs) as defined in Section 2.2. In particular, the set C of elements is the set $\Psi(\Delta) \cup \Psi(Inv)$ of all simple clock constraints of $\mathcal{A}$, and for every $D \cup I$ with $D \subseteq \Psi(\Delta)$ and $I \subseteq \Psi(Inv)$, the predicate $\mathbf{P}$ is defined as $\mathbf{P}(D \cup I) = 1$ iff the reduction $\mathcal{A}_{<D,I>}$ is sufficient. The monotonicity of $\mathbf{P}$ is witnessed in Proposition 10.2. Hence, $\mathbf{P}_1$-minimal subsets correspond to MSRs and $\mathbf{P}_0$-maximal subsets correspond to maximal insufficient reductions.*

10.2.1 Base Scheme For Computing a Minimum MSR

Algorithm 10.1 shows a high-level scheme of our approach for computing a minimum MSR. The algorithm iteratively identifies an ordered set of MSRs, $|M_1| > |M_2| > \cdots > |M_k|$, such that the last MSR M_k is a minimum MSR. Each of the MSRs, say M_i, is identified in two steps. First, the algorithm finds a *seed*, i.e., a reduction N_i such that

```
   input  : a timed automaton $\mathcal{A} = (L, l_0, C, \Delta, Inv)$
   input  : an unreachable set $L_T \subsetneq L$ of locations of $\mathcal{A}$
   output: a minimum MSR of $\mathcal{A}$ w.r.t. $L_T$
1  $N \leftarrow \mathcal{A}_{<\Psi(\Delta),\Psi(Inv)>}$; $\mathcal{M} \leftarrow \emptyset$; $\mathcal{I} \leftarrow \emptyset$
2  while $N \neq$ null do
3      $M, \mathcal{I} \leftarrow$ shrink($N, \mathcal{I}$)                         // Algorithm 10.2
4      $\mathcal{M} \leftarrow \mathcal{M} \cup \{M\}$
5      $N, \mathcal{I} \leftarrow$ findSeed($M, \mathcal{M}, \mathcal{I}$)                // Algorithm 10.3
6  return $M$
```

Algorithm 10.1: A minimum MSR extraction scheme.

N_i is sufficient and $|N_i| < |M_{i-1}|$. Second, the algorithm *shrinks* N_i into an MSR M_i such that $M_i \sqsubseteq N_i$ (and thus $|M_i| \leqslant |N_i|$). The initial seed N_1 is $\mathcal{A}_{<\Psi(\Delta),\Psi(Inv)>}$, i.e., the reduction that removes all simple clock constraints (which makes all locations of $\mathcal{A}$ trivially reachable). Once there is no sufficient reduction N_i with $|N_i| < |M_{i-1}|$, we know that $M_{i-1} = M_k$ is a minimum MSR.

Note that the algorithm also maintains two auxiliary sets, $\mathcal{M}$ and $\mathcal{I}$, to store all identified MSRs and insufficient reductions, respectively. The two sets are used during the process of finding and shrinking a seed which we describe below.

10.2.2 *Shrinking a Seed*

Our approach for shrinking a seed N into an MSR M is based on the domain agnostic single MSMP extractor we presented in 2.3.3, Algorithm 2.1, extended via two concepts that are specific for MSRs: *critical simple clock constraints* and *reduction cores*.

Definition 10.6 (critical constraint)**.** *Given a sufficient reduction $\mathcal{A}_{<D,I>}$, a simple clock constraint c is* critical *for $\mathcal{A}_{<D,I>}$ iff $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ is insufficient.*

Proposition 10.3. *If a constraint $c \in D \cup I$ is critical for a sufficient reduction $\mathcal{A}_{<D,I>}$ then c is critical for every sufficient reduction $\mathcal{A}_{<D',I'>}$, $\mathcal{A}_{<D',I'>} \sqsubseteq \mathcal{A}_{<D,I>}$. Moreover, by Definitions. 10.5 and 10.6, $\mathcal{A}_{<D,I>}$ is an MSR iff every $c \in D \cup I$ is* critical *for $\mathcal{A}_{<D,I>}$.*

Proof. By contradiction, assume that c is critical for $\mathcal{A}_{<D,I>}$ but not for $\mathcal{A}_{<D',I'>}$, i.e., $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ is insufficient and $\mathcal{A}_{<D'\setminus\{c\},I'\setminus\{c\}>}$ is sufficient. As $\mathcal{A}_{<D',I'>} \sqsubseteq \mathcal{A}_{<D,I>}$, we have $\mathcal{A}_{<D'\setminus\{c\},I'\setminus\{c\}>} \sqsubseteq \mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$. By Proposition 10.2, if the reduction $\mathcal{A}_{<D'\setminus\{c\},I'\setminus\{c\}>}$ is sufficient then $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ is also sufficient. □

Note that critical constraints are a domain specific instance of the general MSMP concept of *critical elements* (Definition 2.8), and Proposition 10.3 is thus a domain specific instance of the general MSMP Observation 2.7.

Definition 10.7 (reduction core)**.** *Let $\mathcal{A}_{<D,I>}$ be a sufficient reduction, ρ a witness run of the sufficiency (i.e. reachability of L_T on $\mathcal{A}_{<D,I>}$), and*

```
1  $X \leftarrow \emptyset$
2  while $(D \cup I) \neq X$ do
3      $c \leftarrow$ pick a simple clock constraint from $(D \cup I)\setminus X$
4      if $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>} \notin \mathcal{I}$ and $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ is sufficient then
5          $\rho \leftarrow$ a witness run of the sufficiency of $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$
6          $\mathcal{A}_{<D,I>} \leftarrow$ the reduction core of $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ w.r.t. $\rho$
7      else
8          $X \leftarrow X \cup \{c\}$
9          $\mathcal{I} \leftarrow \mathcal{I} \cup \{N \in \mathcal{R}_{\mathcal{A}} \mid N \sqsubseteq \mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}\}$
10 return $\mathcal{A}_{<D,I>}, \mathcal{I}$
```

Algorithm 10.2: shrink($\mathcal{A}_{<D,I>}, \mathcal{I}$)

π the path corresponding to ρ. Futhermore, let $M(\pi) = l_0, e_1, \ldots, e_n, l_n$ be the path corresponding to π on the original TA $\mathcal{A}$ defined as in (10.5). The reduction core *of $\mathcal{A}_{<D,I>}$ w.r.t. ρ is the reduction $A_{<D',I'>}$ where $D' = \{(e, \varphi) \mid (e, \varphi) \in D \wedge e = e_i$ for some $1 \leqslant i \leqslant n\}$ and $I' = \{(l, \varphi) \mid (l, \varphi) \in I \wedge l = l_i$ for some $0 \leqslant l \leqslant n\}$.*

Intuitively, the reduction core of $\mathcal{A}_{<D,I>}$ w.r.t. ρ reduces from $\mathcal{A}$ only the simple clock constraints that appear on the witness path in $\mathcal{A}$.

Proposition 10.4. *Let $\mathcal{A}_{<D,I>}$ be a sufficient reduction, ρ the witness of reachability of L_T on $\mathcal{A}_{<D,I>}$, and $\mathcal{A}_{<D',I'>}$ the reduction core of $\mathcal{A}_{<D,I>}$ w.r.t. ρ. Then $\mathcal{A}_{<D',I'>}$ is a sufficient reduction and $\mathcal{A}_{<D',I'>} \sqsubseteq \mathcal{A}_{<D,I>}$.*

Proof. By Definition 10.7, $D' \subseteq D$ and $I' \subseteq I$, thus $\mathcal{A}_{<D',I'>} \sqsubseteq \mathcal{A}_{<D,I>}$. As for the sufficiency of $\mathcal{A}_{<D',I'>}$, we only sketch the proof. Intuitively, both $\mathcal{A}_{<D,I>}$ and $\mathcal{A}_{<D',I'>}$ originate from $\mathcal{A}$ by only removing some simple clock constraints ($D \cup I$, and $D' \cup I'$, respectively), i.e., the graph structure of $\mathcal{A}_{<D,I>}$ and $\mathcal{A}_{<D',I'>}$ is the same, however, some corresponding paths of $\mathcal{A}_{<D,I>}$ and $\mathcal{A}_{<D',I'>}$ differ in the constraints that appear on the paths. By Definition 10.7, the path π that corresponds to the witness run ρ of $\mathcal{A}_{<D,I>}$ is also a path of $\mathcal{A}_{<D',I'>}$. Since realizability of a path depends only on the constraints along the path, if π is realizable on $\mathcal{A}_{<D,I>}$ then π is also realizable on $\mathcal{A}_{<D',I'>}$. □

Our approach for shrinking a sufficient reduction N is shown in Algorithm 10.2. The algorithm iteratively maintains a sufficient reduction $\mathcal{A}_{<D,I>}$ and a set X of known critical constraints for $\mathcal{A}_{<D,I>}$. Initially, $\mathcal{A}_{<D,I>} = N$ and $X = \emptyset$. In each iteration, the algorithm picks a simple clock constraint $c \in (D \cup I)\setminus X$ and checks the reduction $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ for sufficiency. If $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ is insufficient, the algorithm adds c to X. Otherwise, if $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ is sufficient, the algorithm obtains a witness run ρ of the sufficiency from the verifier and reduces $\mathcal{A}_{<D,I>}$ to the corresponding reduction core. The algorithm terminates when $(D \cup I) = X$. An invariant of the algorithm is that every $c \in X$ is critical for $\mathcal{A}_{<D,I>}$. Thus, when $(D \cup I) = X$, $\mathcal{A}_{<D,I>}$ is an MSR (Proposition 10.3).

```
1 while $\{N \in \mathcal{R}_{\mathcal{A}} \mid N \notin \mathcal{I} \wedge \forall M' \in \mathcal{M}. N \not\sqsubseteq M' \wedge |N| = |M| - 1\} \neq \emptyset$ do
2     $N \leftarrow$ pick from $\{N \in \mathcal{R}_{\mathcal{A}} \mid N \notin \mathcal{I} \wedge \forall M' \in \mathcal{M}. N \not\sqsubseteq M' \wedge |N| = |M| - 1\}$
3     if $N$ is sufficient then return $N, \mathcal{I}$
4     else $\mathcal{I} \leftarrow \mathcal{I} \cup \{N' \in \mathcal{R}_{\mathcal{A}} \mid N' \sqsubseteq \mathsf{enlarge}(N)\}$        // Algorithm 10.4
5 return null, $\mathcal{I}$
```

Algorithm 10.3: findSeed$(M, \mathcal{M}, \mathcal{I})$

Note that the algorithm also uses the set $\mathcal{I}$ of known insufficient reductions. In particular, before calling a verifier to check a reduction for sufficiency (line 4), the algorithm first checks (in a lazy manner) whether the reduction is already known to be insufficient. Also, whenever the algorithm determines a reduction $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ to be insufficient, it adds $\mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ and every N, $N \sqsubseteq \mathcal{A}_{<D\setminus\{c\},I\setminus\{c\}>}$ to $\mathcal{I}$ (by Proposition 10.2, every such N is also insufficient).

10.2.3 *Finding a Seed*

We now describe the procedure findSeed. The input is the last identified MSR M, the set $\mathcal{M}$ of known MSRs, and the set $\mathcal{I}$ of known insufficient reductions. The output is a seed, i.e., a sufficient reduction N such that $|N| < |M|$, or `null` if there is no seed. Let us by `CAND` denote the set of all *candidates* on a seed, i.e., `CAND` $= \{N \in \mathcal{R}_{\mathcal{A}} \mid |N| < |M|\}$. A brute-force approach would be to check individual reductions in `CAND` for sufficiency until a sufficient one is found, however, this can be practically intractable since $|\texttt{CAND}| = \sum_{i=1}^{|M|} \binom{|\Psi(\Delta) \cup \Psi(Inv)|}{i-1}$.

We provide three observations to prune the set `CAND` of candidates that need to be tested for being a seed. The first observation exploits the set $\mathcal{I}$ of already known insufficient reductions: no $N \in \mathcal{I}$ can be a seed. The second observation exploits the set $\mathcal{M}$ of already known MSRs. By the definition of an MSR, for every $M' \in \mathcal{M}$ and every N such that $N \sqsubsetneq M'$, the reduction N is necessarily insufficient and hence cannot be a seed. The third observation is the following:

Proposition 10.5. *For every sufficient reduction $N \in$ `CAND` there exists a sufficient reduction $N' \in$ `CAND` such that $N \sqsubseteq N'$ and $|N'| = |M| - 1$.*

Proof. If $|N| = |M| - 1$, then $N = N'$. For the other case, when $|N| < |M| - 1$, let $N = \mathcal{A}_{<D^N,I^N>}$ and $M = \mathcal{A}_{<D^M,I^M>}$. We construct $N' = \mathcal{A}_{<D^{N'},I^{N'}>}$ by adding arbitrary $(|M| - |N|) - 1$ simple clock constraint from $(D^M \cup I^M) \setminus (D^N \cup I^N)$ to $(D^N \cup I^N)$, i.e., $D^N \cup I^N \subseteq D^{N'} \cup I^{N'} \subseteq (D^M \cup I^M \cup D^N \cup I^N)$ and $|D^{N'} \cup I^{N'}| = |M| - 1$. By definition of `CAND`, $N' \in$ `CAND`. Moreover, since $N \sqsubsetneq N'$ and N is sufficient, then N' is also sufficient (Proposition 10.2). □

Based on the above observations, we build a set $\mathcal{C}$ of indispensable candidates on seeds that need to be tested for sufficiency:

$$\mathcal{C} = \{N \in \mathcal{R}_{\mathcal{A}} \mid N \notin \mathcal{I} \wedge \forall M' \in \mathcal{M}. N \not\sqsubseteq M' \wedge |N| = |M| - 1\} \quad (10.6)$$

The procedure findSeed is shown in Algorithm 10.3. In each iteration, the algorithm picks a reduction $N \in \mathcal{C}$ and checks it for

```
1 foreach $c \in (\Psi(\Delta) \cup \Psi(Inv)) \setminus (D \cup I)$ do
2     if $c \in \Psi(\Delta)$ and $\mathcal{A}_{<D\cup\{c\},I>}$ is sufficient then $D \leftarrow D \cup \{c\}$
3     if $c \in \Psi(Inv)$ and $\mathcal{A}_{<D,I\cup\{c\}>}$ is sufficient then $I \leftarrow I \cup \{c\}$
4 return $\mathcal{A}_{<D,I>}$
```

Algorithm 10.4: enlarge($\mathcal{A}_{<D,I>}$)

sufficiency (via the verifier). If N is sufficient, the procedure returns N as the seed. In the other case, when N is insufficient, the algorithm first *enlarges* N into an insufficient reduction E such that $N \sqsubseteq E$. By Proposition 10.2, every reduction N' such that $N' \sqsubseteq E$ is also insufficient, thus all these reductions are added to $\mathcal{I}$ and hence removed from $\mathcal{C}$ (note that this includes also N). If $\mathcal{C}$ becomes empty, then there is no seed.

The purpose of *enlarging* N into E is to quickly prune the candidate set $\mathcal{C}$. We could just add all the insufficient reductions $\{N' \mid N' \sqsubseteq N\}$ to $\mathcal{I}$, but note that $|\{N' \mid N' \sqsubseteq E\}|$ is exponentially larger than $|\{N' \mid N' \sqsubseteq N\}|$ w.r.t. $|E| - |N|$. We describe the *enlargement* of N in Algorithm 10.4. Let $N = \mathcal{A}_{<D,I>}$. The algorithm attempts to one by one add the constraints from $\Psi(\Delta) \setminus D$ and $\Psi(Inv) \setminus I$ to D and I, respectively, checking each emerged reduction for sufficiency, and keeping only the changes that preserve $\mathcal{A}_{<D,I>}$ to be insufficient. Note that the algorithm is based on the domain agnostic MSMP growing procedure presented in Section 2.3.3, Algorithm 2.2.

10.2.4 *Representation of $\mathcal{I}$ and $\mathcal{C}$*

The final piece of the puzzle is how to efficiently manipulate with the sets $\mathcal{I}$ and $\mathcal{C}$. In particular, we are adding reductions to $\mathcal{I}$ and $\mathcal{C}$, removing reductions from $\mathcal{C}$, checking if a reduction belongs to $\mathcal{I}$, checking if $\mathcal{C}$ is empty, and picking a reduction from $\mathcal{C}$. The problem is that the size of these sets can be exponential w.r.t. $|\Psi(\Delta) \cup \Psi(Inv)|$ (there are exponentially many reductions), and thus, it is practically intractable to maintain the sets explicitly. Instead, we use a symbolic representation that is based on the representation of *unexplored subsets* via the formula $map^+ \wedge map^-$ (Section 2.3.2). Given a TA $\mathcal{A}$ with simple clock constraints $\Psi(\Delta) = \{(e_1, \varphi_1), \dots, (e_p, \varphi_p)\}$ and $\Psi(Inv) = \{(l_1, \varphi_1), \dots, (l_q, \varphi_q)\}$, we introduce two sets $X = \{x_1, \dots, x_p\}$ and $Y = \{y_1, \dots, y_q\}$ of Boolean variables. Note that every valuation of the variables $X \cup Y$ one-to-one maps to the reduction $\mathcal{A}_{<D,I>}$ such that $(e_i, \varphi_i) \in D$ iff x_i is assigned *True* and $(l_j, \varphi_j) \in I$ iff y_j is assigned *True*.

The set $\mathcal{I}$ is gradually build during the whole computation of Algorithm 10.1. To represent the set $\mathcal{I}$, we build a Boolean formula $\mathbb{I}$ such that a reduction N belongs to $\mathcal{I}$ iff N **does not** correspond to a model model of $\mathbb{I}$. Initially, $\mathcal{I} = \emptyset$, thus $\mathbb{I} = True$. To add an insufficient reduction $\mathcal{A}_{<D,I>}$ and all reductions N, $N \sqsubseteq \mathcal{A}_{<D,I>}$, to $\mathcal{I}$, we add to $\mathbb{I}$ the clause $(\bigvee_{(e_i,\varphi_i) \in \Psi(\Delta) \setminus D} x_i) \vee (\bigvee_{(l_j,\varphi_j) \in \Psi(Inv) \setminus I} y_j)$.

To test if a reduction N is in the set $\mathcal{I}$, we check if the valuation of $X \cup Y$ that corresponds to N is not a model of $\mathcal{I}$.

The set $\mathcal{C}$ is repeatedly build during each call of the procedure findSeed based on Eq. 10.6 and it is encoded via a Boolean formula $\mathbb{C}$ such that every model of $\mathbb{C}$ **does** correspond to a reduction $N \in \mathcal{C}$:

$$\mathbb{C} = \mathbb{I} \wedge \bigwedge_{\mathcal{A}_{<D,I>} \in \mathcal{M}} \left(\left(\bigvee_{(e_i,\varphi_i) \in \Psi(\Delta) \setminus D} x_i \right) \vee \left(\bigvee_{(l_j,\varphi_j) \in \Psi(Inv) \setminus I} y_j \right) \right) \wedge \mathtt{trues}(|\mathtt{M}| - 1) \tag{10.7}$$

where $\mathtt{trues}(|\mathtt{M}| - 1)$ is a cardinality encoding enforcing that exactly $|M| - 1$ variables from $X \cup Y$ are set to *True*. To check if $\mathcal{C} = \emptyset$ or to pick a reduction $N \in \mathcal{C}$, we ask a SAT solver for a model of $\mathbb{C}$. To remove an insufficient reduction from $\mathcal{C}$, we update the formula $\mathbb{I}$ (and thus also $\mathbb{C}$) as described above.

10.2.5 *Related Work*

The concept of minimal sufficient reductions (MSRs) is novel in the context of timed automata, and hence we proposed the first algorithm tailored to the identification of a minimum MSR. However, since MSRs are an instance of minimal sets over a monotone predicate, there exist many MSMP identification/enumeration algorithms that we could also apply. Perhaps the simplest solution is to use a domain agnostic MSMP enumeration algorithm (e.g. the algorithms MARCO, ReMUS and TOME discussed in Chapter 4) to enumerate all MSRs, and then select a one with the minimum cardinality. However, since there can be up to exponentially many MSRs, this approach might be very inefficient or even practically intractable.

There have been proposed several algorithms for extracting a minimum MSMP for particular instances of MSMPs, mainly for extracting a minimum minimal unsatisfiable subset (MUS) of a given CNF formula [Ignatiev et al., 2015, Liffiton et al., 2009, Ignatiev et al., 2016]. All these algorithms can be in some extent generalized and applied also for finding a minimum MSR. However, since the algorithms are dedicated to the particular instances of MSMP and extensively exploit specific properties of the instances (such as we exploit reduction cores in case of MSRs), they would be rather inefficient for our use case. Worth of a more detailed discussion is an algorithm [Ignatiev et al., 2015] for extraction of a minimum MUS that is based on the minimal hitting set duality between MUSes and minimal correction subsets (MCSes). The algorithm gradually maintains a set *kMCSes* of known MCSes; initially *kMCes* $= \emptyset$. In each iteration, the algorithm computes a minimum minimal hitting H set of *kMCSes* and checks H for satisfiability. If H is unsatisfiable then it is guaranteed to be a minimum MUS and the algorithm terminates. Otherwise, H is grown to a maximal satisfiable subset whose complement is an MCS, thus *kMCSes* is enlarged, and the algorithm continues with a next iteration. Since this duality between MUSes and MCSes applies for MSMPs in general (Observation 2.6), the base idea of their algorithm can be straightforwardly used also for extracting a minimum MSR. However, the disadvantage of their approach is that they either identify the minimum MUS (MSR) within a given time limit or they identify no MUS (MSR) at all. On the other hand, our algorithm gradually identifies a sequence of MSRes such that each identified MSR is smaller than the previous one. Hence, if finding a minimum MSR is practically intractable, we can often provide at least a non-minimum MSR, which can still be used to relax the input TA $\mathcal{A}$ in a reasonable way.

10.3 Synthesis of Relaxation Parameters

The main objective of this study is to make the target locations L_T of a given TA $\mathcal{A} = (L, l_0, C, \Delta, Inv)$ reachable by only modifying the constants of simple constraints of $\mathcal{A}$. In the previous section, we presented an efficient algorithm to find a set of simple clock constraints $D \subseteq \Psi(\Delta)$ (10.1) (over transitions) and $I \subseteq \Psi(Inv)$ (10.2) (over locations) such that the target set is reachable when constraints D and I are removed from $\mathcal{A}$. In other words, L_T is reachable on $\mathcal{A}_{<D,I>}$. Consequently, a verifier (model-checker) generates a finite run

$$\rho'_{L_T} = (l_0, \mathbf{0}) \rightarrow_{d_0} (l_1, v_1) \rightarrow_{d_1} \ldots \rightarrow_{d_{n-1}} (l_n, v_n)$$

of $\mathcal{A}_{<D,I>}$ such that $l_n \in L_T$. Let

$$\pi'_{L_T} = l_0, e'_1, l_1, \ldots, e'_{n-1}, l_n$$

be the corresponding path of $\mathcal{A}_{<D,I>}$, i.e., the path π'_{L_T} is realizable on $\mathcal{A}_{<D,I>}$ due to the delay sequence $d_0, d_1, \ldots, d_{n-1}$ and the resulting run is ρ'_{L_T}. The corresponding path on the original TA $\mathcal{A}$ defined as in (10.5) is the following:

$$\pi'_{L_T} = M(\pi_{L_T}), \text{ and } \pi_{L_T} = l_0, e_1, l_1, \ldots, e_{n-1}, l_n \qquad (10.8)$$

While π'_{L_T} is realizable on $\mathcal{A}_{<D,I>}$, π_{L_T} is not realizable on $\mathcal{A}$ since L_T is not reachable on $\mathcal{A}$. We present a MILP based method to find a relaxation valuation $\mathbf{r} : D \cup I \rightarrow \mathbb{N} \cup \{\infty\}$ such that the path induced by π_{L_T} is realizable on $\mathcal{A}_{<D,I,\mathbf{r}>}$.

For a given automaton path $\pi = l_0, e_1, l_1, \ldots, e_{n-1}, l_n$ with $e_i = (l_{i-1}, \lambda_i, \phi_i, l_i)$ for each $i = 1, \ldots, n-1$, we introduce real valued delay variables $\delta_0, \ldots, \delta_{n-1}$ that represent the time spent in each location along the path. Since clocks measure the time passed since their last resets, for a fixed path, a clock on a given constraint (invariant or guard) can be mapped to a sum of delay variables:

$$\begin{aligned} \Gamma(x, \pi, i) &= \delta_k + \delta_{k+1} + \ldots + \delta_{i-1} \\ &\quad \text{where } k = \max(\{m \mid x \in \lambda_m, m < i\} \cup \{0\}) \end{aligned} \qquad (10.9)$$

The value of clock x equals to $\Gamma(x, \pi, i)$ on the i-th transition e_i along π. In (10.9), k is the index of the transition where x is last reset before e_i along π, and it is 0 if it is not reset. $\Gamma(0, \pi, i)$ is defined as 0 for notational convenience.

Guards. For transition e_i, each simple constraint $\varphi = x - y \sim c \in \mathcal{S}(\phi_i)$ on the guard ϕ_i is mapped to the new delay variables as:

$$\Gamma(x, \pi, i) - \Gamma(y, \pi, i) \sim c + p_{e_i,\varphi} \tag{10.10}$$

where $p_{e_i,\varphi}$ is a new integer valued relaxation variable if $(e_i, \varphi) \in D$, otherwise it is set to 0.

Invariants. Each simple clock constraint $\varphi = x - y \sim c \in \mathcal{S}(Inv(l_i))$ of the invariant of location l_i is mapped to arriving (10.11) and leaving (10.12) constraints over the delay variables, since the invariant should be satisfied when arriving (i.e. lower bounds) and leaving (i.e. upper bounds) the location:

$$\Gamma(x, \pi, i) \cdot \mathbf{I}(x \notin \lambda_i) - \Gamma(y, \pi, i) \cdot \mathbf{I}(y \notin \lambda_i) \sim c + p_{l_i,\varphi_i} \quad \textit{if } i > 0 \tag{10.11}$$

$$\Gamma(x, \pi, i+1) - \Gamma(y, \pi, i+1) \sim c + p_{l_i,\varphi_i} \tag{10.12}$$

where $\mathbf{I}$ is a binary function mapping *true* to 1 and *false* to 0, p_{l_i,φ_i} is a new integer valued variable if $(l_i, \varphi_i) \in I$, otherwise it is set to 0.

Finally, we define a MILP (10.13) for the path π. The constraint relaxation variables $\{p_{l,\varphi} \mid (l, \varphi) \in I\}$ and $\{p_{e,\varphi} \mid (e, \varphi) \in D\}$ (integer valued), and the delay variables $\delta_0, \dots, \delta_{n-1}$ (real valued) are the decision variables of the MILP.

$$\begin{aligned} \text{minimize} \quad & \sum_{(l,\varphi)\in I} p_{l,\varphi} + \sum_{(e,\varphi)\in D} p_{e,\varphi} \\ \text{subject to} \quad & (10.10) \text{ for each } i = 1, \dots, n-1, \text{ and } x - y \sim c \in \mathcal{S}(\phi_i) \\ & (10.11) \text{ for each } i = 1, \dots, n, \text{ and } x - y \sim c \in \mathcal{S}(Inv(l_i)) \\ & (10.12) \text{ for each } i = 0, \dots, n-1, \text{ and } x - y \sim c \in \mathcal{S}(Inv(l_i)) \\ & p_{l,\varphi} \in \mathbb{Z}_+ \text{ for each } (l, \varphi) \in I, \\ & \text{and } \; p_{e,\varphi} \in \mathbb{Z}_+ \text{ for each } (e, \varphi) \in D \end{aligned} \tag{10.13}$$

Let $\{p^\star_{l,\varphi} \mid (l, \varphi) \in I\}$, $\{p^\star_{e,\varphi} \mid (e, \varphi) \in D\}$, and $\delta^\star_0, \dots, \delta^\star_{n-1}$ denote the solution of MILP (10.13) when one exists. We define a relaxation valuation $\mathbf{r}$ with respect to the solution as

$$\mathbf{r}(l, \varphi) = p^\star_{l,\varphi} \text{ for each } (l, \varphi) \in I, \quad \mathbf{r}(e, \varphi) = p^\star_{e,\varphi} \text{ for each } (e, \varphi) \in D. \tag{10.14}$$

Proposition 10.6. *Let $\mathcal{A} = (L, l_0, C, \Delta, Inv)$ be a timed automaton, $\pi = l_0, e_1, l_1, \dots, e_n, l_n$ be a finite path of $\mathcal{A}$, and $D \subseteq \Psi(\Delta)$, $I \subseteq \Psi(I)$ be guard and invariant constraint sets. If the MILP constructed from $\mathcal{A}$, π, D and I as defined in (10.13) is feasible, then l_n is reachable on $\mathcal{A}_{<D,I,\mathbf{r}>}$ with $\mathbf{r}$ as defined in (10.14).*

Proof. Let us denote the optimal solution of MILP (10.13) by $\{p^\star_{l,\varphi} \mid (l, \varphi) \in I\}$, $\{p^\star_{e,\varphi} \mid (e, \varphi) \in D\}$, and $\delta^\star_0, \dots, \delta^\star_{n-1}$. For simplicity of presentation, let us set $p^\star_{l,\varphi}$ to 0 for each $(l, \varphi) \in \Psi(Inv) \backslash I$ and set $p^\star_{e,\varphi}$ to 0 for each $(e, \varphi) \in \Psi(\Delta) \backslash D$. Let $\mathcal{A}_{<D,I,\mathbf{r}>} = (L, l_0, C, \Delta', Inv')$ and $T(\mathcal{A}_{<D,I,\mathbf{r}>}) = (S, s_0, \Sigma, \rightarrow)$. We define clock value sequence

$v_0, v_1, \ldots, v_n$ with respect to the path π with $e_i = (l_{i-1}, \lambda_i, \phi_i, l_i)$ and the delay sequence $\delta_0^\star, \ldots, \delta_{n-1}^\star$ iteratively as $v_0 = \mathbf{0}$ and $v_i = (v_{i-1} + \delta_{i-1}^\star)[\lambda_i := 0]$ for each $i = 1, \ldots, n$. Along the path π, v_i is consistent with $\Gamma(\cdot, \pi, i)$ (10.9) such that

$$a)\ v_i(x) = \Gamma(x, \pi, i).I(x \notin \lambda_i) \qquad \textit{and} \quad b)\ v_i(x) + \delta_i^\star = \Gamma(x, \pi, i+1) \tag{10.15}$$

For a simple constraint $\varphi = x - y \sim c + p_{l_i,\varphi}^\star \in Inv'(l_i)$ (i.e. $x - y \sim c \in Inv(l_i)$ via Definition. 10.4 and (10.14)), it holds that $v_i(x) - v_i(y) \sim c + p_{l_i,\varphi}^\star$ via (10.11) and (10.15)-*a*. Then by (10.14) $v_i \models Inv'(l_i)$ and $(l_i, v_i) \in S$. Similarly, $v_i + \delta_i^\star \models Inv'(l_i)$ via (10.12) and (10.15)-*b*. Hence, $(l_i, v_i + \delta_i^\star) \in S$ and $(l_i, v_i) \xrightarrow{\delta_i^\star} (l_i, v_i + \delta_i^\star)$ (delay transition). Furthermore, by (10.10), (10.14), (10.15)-*b* and Definition. 10.4, we have $v_i + \delta_i^\star \models R(\phi_i, D|_{e_i}, \mathbf{r}|_{e_i})$ and $(l_i, v_i + \delta_i^\star) \xrightarrow{act} (l_{i+1}, v_{i+1})$ (discrete transition). As $s_0 = (l_0, \mathbf{0}) \in S$, and the derivation applies to each $i = 1, \ldots, n$, we reach that $\rho = (l_0, v_0), \ldots, (l_n, v_n) \in [[\mathcal{A}_{<D,I,\mathbf{r}>}]]$, and l_n is reachable on $\mathcal{A}_{<D,I,\mathbf{r}>}$. □

Note that in [Bouyer et al., 2007], a linear programming (LP) based approach was used to generate the optimal delay sequence for a given path of a weighted timed automata. In our case, the optimization problem is in MILP form since we find an integer valued relaxation valuation ($\mathbf{r}$) in addition to the delay variables.

Recall that we construct relaxation sets D and I via Algorithm 10.1, and define π_{L_T} (10.8) that reach L_T such that the corresponding path π'_{L_T} is realizable on $\mathcal{A}_{<D,I>}$. Then, we define MILP (10.13) with respect to π_{L_T}, D and I, and define $\mathbf{r}$ (10.14) according to the optimal solution. Note that this MILP is always feasible since π'_{L_T} is realizable on $\mathcal{A}_{<D,I>}$. Finally, by Proposition 10.6, we conclude that L_T is reachable on $\mathcal{A}_{<D,I,\mathbf{r}>}$.

Example 10.3. *For the TA shown in Figure 10.1, Algorithm 10.1 generates $A_{<D,I>}$ with $D = \{(e_5, x \geqslant 25)\}$ and $I = \{(l_3, u \leqslant 26)\}$ such that $\pi = l_0, e_1, l_1, e_2, l_2, e_3, l_1, e_4, l_3, e_5, l_4$ is realizable on $A_{<D,I>}$. The MILP is constructed for π, D and I with decision variables $p_{e_5,x\geqslant 25}$, $p_{l_3,u\leqslant 26}$, $\delta_0, \delta_1, \delta_2, \delta_3, \delta_4$ and δ_5 as in (10.13). The solution is $p_{e_5,x\geqslant 25} = 3$, $p_{l_3,u\leqslant 26} = 5$, and the delay sequence is $9, 4, 0, 9, 9, 0$. Consequently, l_4 is reachable on $A_{<D,I,\mathbf{r}>}$ with $\mathbf{r}(e_5, x \geqslant 25) = 3$ and $\mathbf{r}(l_3, u \leqslant 26) = 5$.*

10.4 Case Study

We implemented the proposed reduction and relaxation methods in a tool called Tamus. We use UPPAAL for sufficiency checks and witness computation, and CBC solver from Or-tools library[2] for the MILP part. All experiments were run on a laptop with Intel i5 quad core processor at 2.5 GHz and 8 GB ram. The tool and used benchmarks are available at:

https://github.com/jar-ben/tamus

[2] https://developers.google.com/optimization

As discussed in Section 10, an alternative approach to solve our problem (Problem 10.1) is to parameterize each simple clock con-

Model	$\|\Psi\|$	d	v	t	c_m	Model	$\|\Psi\|$	d	v	t	c_m	Model	$\|\Psi\|$	d	v	t	c_m
$\mathcal{A}_{(3,1,12)}$	11	2	33	0.18	6	$\mathcal{A}_{(5,1,12)}$	16	3	120	0.63	10	$\mathcal{A}_{(7,1,12)}$	19	3	120	0.63	11
$\mathcal{A}_{(3,2,12)}$	17	1	13	0.13	13	$\mathcal{A}_{(5,2,12)}$	24	1	42	0.35	13	$\mathcal{A}_{(7,2,12)}$	28	1	95	0.72	13
$\mathcal{A}_{(3,1,18)}$	16	3	61	0.37	9	$\mathcal{A}_{(5,1,18)}$	23	4	149	0.90	16	$\mathcal{A}_{(7,1,18)}$	28	5	313	1.87	25
$\mathcal{A}_{(3,2,18)}$	24	1	40	0.40	6	$\mathcal{A}_{(5,2,18)}$	35	1	57	0.58	6	$\mathcal{A}_{(7,2,18)}$	42	1	70	0.74	6
$\mathcal{A}_{(3,1,24)}$	21	4	97	0.65	12	$\mathcal{A}_{(5,1,24)}$	31	6	327	2.16	24	$\mathcal{A}_{(7,1,24)}$	38	7	709	4.76	35
$\mathcal{A}_{(3,2,24)}$	32	1	80	0.85	16	$\mathcal{A}_{(5,2,24)}$	47	2	169	1.80	31	$\mathcal{A}_{(7,2,24)}$	57	2	201	2.21	21
$\mathcal{A}_{(3,1,30)}$	26	5	141	1.05	15	$\mathcal{A}_{(5,1,30)}$	39	7	541	4.17	31	$\mathcal{A}_{(7,1,30)}$	48	10	1680	14.12	47
$\mathcal{A}_{(3,2,30)}$	40	1	65	0.84	9	$\mathcal{A}_{(5,2,30)}$	59	2	330	3.95	14	$\mathcal{A}_{(7,2,30)}$	72	2	403	5.01	14

Table 10.1: Results for the scheduler TA, where $|\Psi| = |\Psi(\Delta) \cup \Psi(I)|$ is the total number of constraints, $d = |D \cup U|$ is the minimum MSR size, v is the number of reachability checks, t is the computation time in seconds, and c_m is the optimal cost of (10.13).

straint of the TA. Then, we can run a parameter synthesis tool on the parameterized TA to identify the set of all possible valuations of the parameters for which the TA satisfies the reachability property. Subsequently, we can choose the valuations that assign non-zero values (i.e., relax) to the minimum number of parameters, and out of these, we can choose the one with a minimum cumulative change of timing constants. In our experimental evaluation, we evaluate a state-of-the-art parameter synthesis tool called Imitator[André et al., 2012] to run such analysis. Although Imitator is not tailored for our problem, it allows us to measure the relative scalability of our approach compared to a well-established synthesis technique.

We used two collections of benchmarks: one is obtained from the literature, and the other are crafted timed automata modeling a machine scheduling problem. All experiments were run using a time limit of 20 minutes per benchmark. Complete results are available in the online appendix[3].

[3] https://www.fi.muni.cz/~xbendik/phdThesis/

10.4.1 *Machine Scheduling*

A scheduler automaton is composed of a set of paths from location l_0 to location l_1. Each path $\pi = l_0 e_k l_k e_{k+1} \dots l_{k+M-1} e_{k+M} l_1$ represents a particular scheduling scenario where an intermediate location, e.g. l_i for $i = k, \dots, k+M-1$, belongs to a unique path (only one incoming and one outgoing transition). Thus, a TA that has p paths with M intermediate locations in each path has $M \cdot p + 2$ locations and $(M+1) \cdot p$ transitions. Each intermediate location represents a machine operation, and periodic simple clock constraints are introduced to mimic the limitations on the corresponding durations. For example, assume that the total time to use machines represented by locations l_{k+i} and l_{k+i+1} is upper (or lower) bounded by c for $i = 0, 2, \dots, M-2$. To capture such a constraint with a period of $t = 2$, a new clock x is introduced and it is reset and checked on every t^{th} transition along the path, i.e., for every $m \in \{i \cdot t + k \mid i \cdot t \leqslant M-1\}$, let $e_m = (l_m, \lambda_m, \phi_m, l_{m+1})$, add x to λ_m, set $\phi_m := \phi_m \wedge x \leqslant c$ ($x \geqslant c$ for lower bound). A periodic constraint is denoted by $(t, c, \sim)$, where t is its period, c is the timing constant, and $\sim \in \{<, \leqslant, >, \geqslant\}$. A set of such constraints are defined for each path to capture possible restrictions. In addition, a bound T on the total execution time is captured with the constraint $x \leqslant T$ on transition e_{k+M} over a clock x that is not reset on any transition. A realizable path to l_1 represents a feasible scheduling scenario, thus the target set is $L_T = \{l_1\}$. We have generated 24 test cases. A test case $\mathcal{A}_{(c,p,M)}$ represents a timed automaton with $c \in \{3,5,7\}$ clocks, and $p \in \{1,2\}$ paths with $M \in \{12,18,24,30\}$ intermediate locations in each path. $R_{c,i}$ is the set of periodic restrictions defined for the i^{th} path of an automaton with c clocks:

$$\begin{aligned}
R_{3,1} &= \{(2,11,\geqslant),(3,15,\leqslant)\}\\
R_{5,1} &= R_{3,1} \cup \{(4,21,\geqslant),(5,25,\leqslant)\}\\
R_{7,1} &= R_{5,1} \cup \{(6,31,\geqslant),(7,35,\leqslant)\}\\
R_{3,2} &= \{(4,17,\geqslant),(5,20,\leqslant)\}\\
R_{5,2} &= R_{3,2} \cup \{(8,33,\geqslant),(9,36,\leqslant)\}\\
R_{7,2} &= R_{5,2} \cup \{(12,49,\geqslant),(12,52,\leqslant)\}
\end{aligned}$$

Note that $\mathcal{A}_{(c,2,M)}$ emerges from $\mathcal{A}_{(c,1,M)}$ by adding a path with restrictions $R_{c,2}$.

Table 10.1 shows results achieved by Tamus on these machine scheduling models. Tamus solved all models and the hardest one $\mathcal{A}_{(7,1,30)}$ took only 14.12 seconds. As expected, the computation time t increases with the number $|\Psi|$ of simple clock constraints in the model. Moreover, the computation time highly correlates with the size d of the minimum MSR. Especially, if we compare two generic models $\mathcal{A}_{(c,1,M)}$ and $\mathcal{A}_{(c,2,M)}$, although $\mathcal{A}_{(c,2,M)}$ has one more path and more constraints, Tamus is faster on $\mathcal{A}_{(c,2,M)}$ since it quickly converges to the path with smaller MSRs.

Imitator solved $\mathcal{A}_{(3,1,12)}$, $\mathcal{A}_{(3,2,12)}$, $\mathcal{A}_{(3,1,18)}$, and $\mathcal{A}_{(5,1,12)}$ within 0.08, 0.5, 61, and 67 seconds, and timeouted for the other models. In addition, we run Imitator with a flag "counterexample" that terminates the computation when a satisfying valuation is found. The use of this flag reduced the computation time for the afore mentioned cases, and it allowed to solve two more models: $\mathcal{A}_{(3,2,18)}$ and $\mathcal{A}_{(5,2,12)}$. However, using this flag, Imitator often did not provided a solution that minimizes the number of relaxed simple clock constraints.

10.4.2 *Benchmarks from the Literature*

We collected 10 example models from the literature shown in Table 10.2. The examples include models with a safety specification that requires avoiding a set of locations L_A, and models with a reachability specification with a set of target locations L_T as considered in this chapter. In both cases, the original models satisfy the given specification. For the first case, we define L_A as the target set and apply our method. Here, we find the minimal number of timing constants that should be changed to reach L_A, i.e., to violate the original safety specification. As such, the proposed approach can be used to analyze robustness for the safety specifications. For the second case,

Model	Source
accel1000	[André et al., 2018][Hoxha et al., 2014]
CAS	[Aichernig et al., 2013]
coffee	[André et al., 2019c]
Jobshop4	[Abdeddaïm and Maler, 2001]
Pipeline3-3	[Knapik and Penczek, 2012]
RCP	[Collomb-Annichini and Sighireanu, 2001]
SIMOP3	[André et al., 2009]
Fischer	[Hune et al., 2002]
JLR13-3tasks	[Jovanovic et al., 2013][André et al., 2015]
WFAS	[Beneš et al., 2015][Arenis et al., 2014]

Table 10.2: Source of the benchmarks from the literature.

inspired from mutation testing [Aichernig et al., 2013], we change a number of constraints on the original model so that L_T becomes unreachable. Eight of the examples are networks of TAs, and while a network of TAs can be represented as a single product TA and hence our approach can handle it, Tamus currently supports only MSR computation for networks of TA, but not MILP relaxation.

The results are shown in Table 10.3. Tamus computed a minimum MSR for all the models and also provided the MILP relaxation for the non-network models. Note that the MILP part always took only few milliseconds (including models from Table 10.1), thus we believe that it would be also the case for the networks of TAs. The base variant of Imitator that computes the set of all satisfying parameter valuations solved only 4 of the 10 models. When run with the early termination flag, Imitator solved 3 more models, however, as discussed above, the provided solutions might not be optimal. We have also evaluated a combination of Tamus and Imitator. In particular, we first run Tamus to compute a minimum MSR $\mathcal{A}_{<D,I>}$, then parameterized the constraints $D \cup I$, and run Imitator on the parameterized TA. In this case, Imitator solved 9 out of 10 models. Moreover, we have the guarantee that we found the optimal solution: the MSR ensures that we relax the minimum number of simple clock constraints, and Imitator finds all satisfying parameterizations of the constraints hence also the one with minimum cumulative change of timing constants.

10.5 *Summary and Future Work*

In this chapter, we proposed the novel concept of a minimum MSR for a TA, that is a minimum set of simple constraints that needs to be relaxed to satisfy a reachability specification. We developed efficient methods to find a minimum MSR, and presented a MILP based solution to tune these constraints. Our analysis on benchmarks showed that our tool Tamus can generate a minimum MSR within seconds even for large systems. In addition, we compared our results with Imitator and observed that Tamus scales much better. However, Tamus minimizes the cumulative change of the constraints from a minimum MSR by considering a single witness path. If the goal is to find a minimal relaxation globally, i.e., w.r.t. all witness paths for

Model	Spec.	$\lvert\Psi\rvert$	$\lvert\Psi^u\rvert$	d	m	v	t	c_m	t^I	t^{IT}	t^{Ic}	t^{ITc}
accel1000	reachability	7690	13	2	3	22	1.83	-	182.5	2.08	1.77	1.03
CAS	reachability	18	18	2	9	46	0.31	14	0.75	0.11	0.09	0.01
coffee	reachability	10	10	2	3	18	0.07	14	0.008	0.002	0.007	0.003
Jobshop4	reachability	64	48	5	5	272	1.99	-	to	949.5	to	942.3
Pipeline3-3	reachability	41	41	1	12	42	0.37	-	to	0.08	to	0.05
RCP	reachability	48	48	1	11	19	0.41	-	to	0.02	24.23	0.02
SIMOP3	reachability	80	80	6	40	903	10.65	-	to	7.26	to	0.49
Fischer	safety	24	16	1	0	14	0.08	-	to	to	0.21	0.01
JLR13-3tasks	safety	42	36	1	0	40	0.41	-	to	2.60	0.05	0.08
WFAS	safety	32	24	1	0	10	0.08	-	16.20	0.01	0.03	0.006

Table 10.3: Experimental results for the benchmarks, where $|\Psi|$, d, v t and c_m are as defined in Table 10.1, $|\Psi^u|$ is the number of constraints considered in the analysis and m is the number of mutated constraints. t^I, t^{IT}, t^{Ic} and t^{ITc} are the Imitator computation times, where c indicates that the early termination flag ("counterexample") is used, otherwise the largest set of parameters is searched, and T indicates that only the constraints from the MSR identified by Tamus are parametrized, otherwise all constraints from Ψ^u are parametrized. *to* shows that the timeout limit is reached (20 min.).

the MSR, we recommend to use the combined version of Tamus and Imitator, i.e., first run Tamus to find a minimum MSR, parametrize each constraint from the MSR and run Imitator to find all satisfying parameter valuations, including the global optimum.

There are several directions for the future work. One of them is to implement the support for MILP computation over networks of timed automata. Another interesting direction is to examine how our approach applies to robustness analysis. In the experimental evaluation, we have already suggested that MSRs can be also used to identify the minimum set of simple clock constraints that need to be removed from the system to violate a safety property. Dually, one might think of a minimum set of simple clock constraints that need to be left in the system to satisfy a safety property.

Bibliography

Yasmina Abdeddaïm and Oded Maler. Job-shop scheduling using timed automata. In *CAV*, volume 2102 of *LNCS*, pages 478–492. Springer, 2001.

Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In *TAP@STAF*, volume 7942 of *LNCS*, pages 20–38. Springer, 2013.

Rajeev Alur. Timed automata. In *CAV*, volume 1633 of *LNCS*, pages 8–22. Springer, 1999.

Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. CEGAR-based formal hardware verification: A case study. Technical report, University of Michigan, CSE-TR-531-07, 2007.

Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *LPAR*, volume 5330 of *LNCS*, pages 343–352. Springer, 2008.

Étienne André. A benchmark library for parametric timed model checking. In *FTSCS*, volume 1008 of *Communications in Computer and Information Science*, pages 75–83. Springer, 2018.

Étienne André. What's decidable about parametric timed automata? *Int. J. Softw. Tools Technol. Transf.*, 21(2):203–219, 2019.

Étienne André, Thomas Chatain, Olivier De Smet, Laurent Fribourg, and Silvain Ruel. Synthèse de contraintes temporisées pour une architecture d'automatisation en réseau. *Journal Européen des Systèmes Automatisés*, 43, 2009.

Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *LNCS*, pages 33–36. Springer, 2012.

Étienne André, Giuseppe Lipari, Hoang Gia Nguyen, and Youcheng Sun. Reachability preservation based parameter synthesis for timed automata. In *NFM*, volume 9058 of *LNCS*, pages 50–65. Springer, 2015.

Étienne André, Ichiro Hasuo, and Masaki Waga. Offline timed pattern matching under uncertainty. In *ICECCS*, pages 10–20. IEEE Computer Society, 2018.

Étienne André, Paolo Arcaini, Angelo Gargantini, and Marco Radavelli. Repairing timed automata clock guards through abstraction and testing. In *TAP@FM*, volume 11823 of *LNCS*, pages 129–146. Springer, 2019a.

Étienne André, Laurent Fribourg, Jean-Marc Mota, and Romain Soulat. Verification of an industrial asynchronous leader election algorithm using abstractions and parametric model checking. In *VMCAI*, volume 11388 of *LNCS*, pages 409–424. Springer, 2019b.

Étienne André, Michal Knapik, Didier Lime, Wojciech Penczek, and Laure Petrucci. Parametric verification: An introduction. *Trans. Petri Nets Other Model. Concurr.*, 14:64–100, 2019c.

Sergio Feo Arenis, Bernd Westphal, Daniel Dietsch, Marco Muñiz, and Ahmad Siyar Andisha. The wireless fire alarm system: Ensuring conformance to industrial standards through formal verification. In *FM*, volume 8442 of *LNCS*, pages 658–672. Springer, 2014.

M. Fareed Arif, Carlos Mencía, and João Marques-Silva. Efficient axiom pinpointing with EL2MCS. In *KI*, volume 9324 of *LNCS*, pages 225–233. Springer, 2015a.

M. Fareed Arif, Carlos Mencía, and João Marques-Silva. Efficient MUS enumeration of Horn formulae with applications to axiom pinpointing. In *SAT*, volume 9340 of *LNCS*, pages 324–342. Springer, 2015b.

M. Fareed Arif, Carlos Mencía, Alexey Ignatiev, Norbert Manthey, Rafael Peñaloza, and João Marques-Silva. BEACON: an efficient sat-based tool for debugging *EL*+ ontologies. In *SAT*, volume 9710 of *LNCS*, pages 521–530. Springer, 2016.

Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter J. Stuckey. #∃sat: Projected model counting. In *SAT*, volume 9340 of *LNCS*, pages 121–137. Springer, 2015.

Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV (2)*, volume 9207 of *LNCS*, pages 70–86. Springer, 2015.

Fahiem Bacchus and George Katsirelos. Finding a collection of MUSes incrementally. In *CPAIOR*, volume 9676 of *LNCS*, pages 35–44. Springer, 2016.

Fahiem Bacchus, Jessica Davies, Maria Tsimpoukelli, and George Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841. AAAI Press, 2014.

John Backes, Darren D. Cofer, Steven P. Miller, and Michael W. Whalen. Requirements analysis of a quad-redundant flight control system. In *NFM*, volume 9058 of *LNCS*, pages 82–96. Springer, 2015.

James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186. Springer, 2005.

R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI*, pages 276–281. Morgan Kaufmann, 1993.

Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative verification of neural networks and its security applications. In *ACM Conference on Computer and Communications Security*, pages 1249–1264. ACM, 2019.

Jiří Barnat, Petr Bauch, and Luboš Brim. Checking sanity of software requirements. In *SEFM 2012 Proceedings*, volume 7504 of *LNCS*, pages 48–62. Springer, 2012.

Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analysing sanity of requirements for avionics systems. *FAoC*, pages 1–19, 2016.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *AAAI/IAAI*, pages 157–162. AAAI Press / The MIT Press, 2000.

Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *QEST*, pages 125–126. IEEE Computer Society, 2006.

Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, pages 37–40. FMCAD Inc., 2011.

Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 8:123–128, 2012.

Anton Belov, Marijn Heule, and João Marques-Silva. MUS extraction using clausal proofs. In *SAT*, volume 8561 of *LNCS*, pages 48–57. Springer, 2014.

Rachel Ben-Eliyahu and Rina Dechter. On computing minimal models. In *AAAI*, pages 2–8. AAAI Press / The MIT Press, 1993.

Jaroslav Bendík. Consistency checking in requirements analysis. In *ISSTA*, pages 408–411. ACM, 2017.

Jaroslav Bendík and Ivana Černá. Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 131–142. EasyChair, 2018.

Jaroslav Bendík and Ivana Černá. MUST: minimal unsatisfiable subsets enumeration tool. In *TACAS (1)*, volume 12078 of *LNCS*, pages 135–152. Springer, 2020a.

Jaroslav Bendík and Ivana Černá. Rotation based MSS/MCS enumeration. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 120–137. EasyChair, 2020b.

Jaroslav Bendík and Ivana Černá. Replication-guided enumeration of minimal unsatisfiable subsets. In *CP*, volume 12333 of *LNCS*, pages 37–54. Springer, 2020c.

Jaroslav Bendík and Kuldeep S. Meel. Approximate counting of minimal unsatisfiable subsets. In *CAV (1)*, volume 12224 of *LNCS*, pages 439–462. Springer, 2020.

Jaroslav Bendík and Kuldeep S. Meel. Counting maximal satisfiable subsets. In *AAAI*, pages 3651–3660. AAAI Press, 2021.

Jaroslav Bendík, Nikola Beneš, Jiří Barnat, and Ivana Černá. Finding boundary elements in ordered sets with application to safety and requirements analysis. In *SEFM*, volume 9763 of *LNCS*, pages 121–136. Springer, 2016a.

Jaroslav Bendík, Nikola Beneš, Ivana Černá, and Jiří Barnat. Tunable online MUS/MSS enumeration. In *FSTTCS*, volume 65 of *LIPIcs*, pages 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016b.

Jaroslav Bendík, Nikola Beneš, and Ivana Černá. Finding regressions in projects under version control systems. In *ICSOFT*, pages 186–197. SciTePress, 2018a.

Jaroslav Bendík, Ivana Černá, and Nikola Beneš. Recursive online enumeration of all minimal unsatisfiable subsets. In *ATVA*, volume 11138 of *LNCS*, pages 143–159. Springer, 2018b.

Jaroslav Bendík, Elaheh Ghassabani, Michael W. Whalen, and Ivana Černá. Online enumeration of all minimal inductive validity cores. In *SEFM*, volume 10886 of *LNCS*, pages 189–204. Springer, 2018c.

Jaroslav Bendík, Ahmet Sencan, Ebru Aydin Gol, and Ivana Cerná. Timed automata relaxation for reachability. In *TACAS (1)*, volume 12651 of *LNCS*, pages 291–310. Springer, 2021.

Nikola Beneš, Peter Bezděk, Kim Guldstrand Larsen, and Jiří Srba. Language emptiness of continuous-time parametric timed automata. In *ICALP (2)*, volume 9135 of *LNCS*, pages 69–81. Springer, 2015.

Peter Bezděk, Nikola Beneš, Jiří Barnat, and Ivana Černá. LTL parameter synthesis of parametric timed automata. In *SEFM*, volume 9763 of *LNCS*, pages 172–187. Springer, 2016.

Peter Bezděk, Nikola Beneš, Ivana Černá, and Jiří Barnat. On clock-aware LTL parameter synthesis of timed automata. *J. Log. Algebraic Methods Program.*, 99:114–142, 2018.

Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. *Proc. of SAT Competition*, pages 13–14, 2018.

Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. Scalable approximation of quantitative information flow in programs. In *VMCAI*, volume 10747 of *LNCS*, pages 71–93. Springer, 2018.

Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *J. Artif. Intell. Res.*, 10: 457–477, 1999.

Bernard Blackham, Mark H. Liffiton, and Gernot Heiser. Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets. In *RTAS*, pages 169–178. IEEE Computer Society, 2014.

Béla Bollobás, Christian Borgs, Jennifer T. Chayes, Jeong Han Kim, and David Bruce Wilson. The scaling window of the 2-SAT transition. *Random Struct. Algorithms*, 18(3):201–256, 2001.

Patricia Bouyer, Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. On the optimal reachability problem of weighted timed automata. *Formal Methods in System Design*, 31:135–175, 2007.

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *CAV*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *CP*, volume 8124 of *LNCS*, pages 200–216. Springer, 2013.

Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *AAAI*, pages 1722–1730. AAAI Press, 2014.

Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *IJCAI*, pages 3569–3576. IJCAI/AAAI Press, 2016.

Huan Chen and João Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC*, pages 142–147. IEEE, 2011.

Zhi-Zhong Chen and Seinosuke Toda. The complexity of selecting maximal solutions. *Inf. Comput.*, 119(2):231–239, 1995.

John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS J. Comput.*, 3(2): 157–168, 1991.

Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for formal verification. *Int. J. Softw. Tools Technol. Transf.*, 8 (4-5):373–386, 2006.

Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.

Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *JAIR*, 40:701–728, 2011.

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59. IEEE, 2012.

Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

Orly Cohen, Moran Gordon, Michael Lifshits, Alexander Nadel, and Vadim Ryvchin. Designers work less with quality formal equivalence checking. In *Design and Verification Conference (DVCon)*. Citeseer, 2010.

Aurore Collomb-Annichini and Mihaela Sighireanu. Parameterized reachability analysis of the IEEE 1394 root contention protocol using TReX. 2001.

Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. Model-based framework for schedulability analysis using UPPAAL 4.1. *Model-based design for embedded systems*, 1(1):93–119, 2009.

Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

Maria J. García de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *PPDP*, pages 32–43. ACM, 2003.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

J. L. de Siqueira N. and Jean-Francois Puget. Explanation-based generalisation of failures. In *ECAI*, pages 339–344. Pitmann Publishing, London, 1988.

Arnaud Durand, Miki Hermann, and Phokion G. Kolaitis. Subtractive reductions and complete problems for counting complexity classes. *Theor. Comput. Sci.*, 340(3):496–513, 2005.

Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In *ATVA*, volume 9938 of *LNCS*, pages 122–129, 2016.

Bruno Dutertre and Leonardo De Moura. The yices SMT solver. *Tool paper at* `http://yices.csl.sri.com/tool-paper.pdf`, 2(2):1–2, 2006.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15. ACM, 1998.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134. FMCAD Inc., 2011.

E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC*, pages 169–180. ACM, 1982.

Ansgar Fehnker. Scheduling a steel plant with timed automata. In *RTCSA*, pages 280–286. IEEE Computer Society, 1999.

Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012.

Andrew Gacek, John Backes, Mike Whalen, Lucas G. Wagner, and Elaheh Ghassabani. The JKind model checker. In *CAV (2)*, volume 10982 of *LNCS*, pages 20–27. Springer, 2018.

Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient generation of inductive validity cores for safety properties. In *SIGSOFT FSE*, pages 314–325. ACM, 2016.

Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats Per Erik Heimdahl, and Lucas G. Wagner. Proof-based coverage metrics for formal verification. In *ASE*, pages 194–199. IEEE Computer Society, 2017a.

Elaheh Ghassabani, Michael W. Whalen, and Andrew Gacek. Efficient generation of all minimal inductive validity cores. In *FMCAD*, pages 31–38. IEEE, 2017b.

Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Near-uniform sampling of combinatorial spaces using XOR constraints. In *NIPS*, pages 481–488. MIT Press, 2006.

Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. An experimentally efficient method for (MSS, CoMSS) partitioning. In *AAAI*, pages 2666–2673. AAAI Press, 2014.

Éric Grégoire, Yacine Izza, and Jean-Marie Lagniez. Boosting mcses enumeration. In *IJCAI*, pages 1309–1315. ijcai.org, 2018.

Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *SEUS*, volume 4761 of *LNCS*, pages 263–272. Springer, 2007.

Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. Minimal unsatisfiable core extraction for SMT. In *FMCAD*, pages 57–64. IEEE, 2016.

George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with SMT-based techniques. In *FMCAD*, pages 1–9. IEEE, 2008.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.

Fred Hemery, Christophe Lecoutre, Lakhdar Sais, and Frédéric Boussemart. Extracting MUCs from constraint networks. In *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 113–117. IOS Press, 2006.

Thomas A. Henzinger, Joerg Preussig, and Howard Wong-Toi. Some lessons from the HYTECH experience. In *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, volume 3, pages 2887–2892 vol.3, 2001.

Aimin Hou. A theory of measurement in diagnosis from first principles. *AI*, 65(2):281–328, 1994.

Bardh Hoxha, Houssam Abbas, and Georgios E. Fainekos. Benchmarks for temporal logic requirements for automotive systems. In *ARCH@CPSWeek*, volume 34 of *EPiC Series in Computing*, pages 25–30. EasyChair, 2014.

Shi-Yu Huang and Kwang-Ting Tim Cheng. *Formal equivalence checking and design debugging*, volume 12. Springer Science & Business Media, 2012.

Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. *J. Log. Algebraic Methods Program.*, 52-53:183–220, 2002.

Anthony Hunter and Sébastien Konieczny. Measuring inconsistency through minimal inconsistent sets. In *KR*, pages 358–366. AAAI Press, 2008.

Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP*, volume 9255 of *LNCS*, pages 173–182. Springer, 2015.

Alexey Ignatiev, Mikoláš Janota, and João Marques-Silva. Quantified maximum satisfiability. *Constraints An Int. J.*, 21(2):277–302, 2016.

Alexey Ignatiev, António Morgado, and João Marques-Silva. PySAT: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *LNCS*, pages 428–437. Springer, 2018.

Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1), 2016.

Dietmar Jannach and Thomas Schmitz. Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Autom. Softw. Eng.*, 23(1):105–144, 2016.

Mikoláš Janota and João Marques-Silva. On deciding MUS membership with QBF. In *CP*, volume 6876 of *LNCS*, pages 414–428. Springer, 2011.

Mikoláš Janota and João Marques-Silva. On the query complexity of selecting minimal sets for monotone predicates. *Artif. Intell.*, 233: 73–83, 2016.

Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *Int. J. Softw. Tools Technol. Transf.*, 16(2):191–213, 2014.

Aleksandra Jovanovic, Didier Lime, and Olivier H. Roux. Integer parameter synthesis for timed automata. In *TACAS*, volume 7795 of *LNCS*, pages 401–415. Springer, 2013.

Aleksandra Jovanovic, Didier Lime, and Olivier H. Roux. Integer parameter synthesis for real-time systems. *IEEE Trans. Software Eng.*, 41(5):445–461, 2015.

Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *AAAI*, pages 1368–1373. AAAI Press / The MIT Press, 2005.

Hans Kleine Büning and Oliver Kullmann. Minimal unsatisfiability and autarkies. In *Handbook of Satisfiability*, volume 185 of *FAIA*, pages 339–401. IOS Press, 2009.

Hans Kleine Büning and Theodor Lettmann. *Propositional logic - deduction and algorithms*, volume 48 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1999.

Michal Knapik and Wojciech Penczek. Bounded model checking for parametric timed automata. *Trans. Petri Nets Other Model. Concurr.*, 5:141–159, 2012.

Martin Kölbl, Stefan Leue, and Thomas Wies. Clock bound repair for timed systems. In *CAV (1)*, volume 11561 of *LNCS*, pages 79–96. Springer, 2019.

Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

Oliver Kullmann. An application of matroid theory to the SAT problem. In *Computational Complexity Conference*, page 116. IEEE Computer Society, 2000a.

Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000b.

Oliver Kullmann and João Marques-Silva. Computing maximal autarkies with few and simple oracle queries. In *SAT*, volume 9340 of *LNCS*, pages 138–155. Springer, 2015.

Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *Int. J. Softw. Tools Technol. Transf.*, 4(2):224–233, 2003.

Orna Kupferman, Wenchao Li, and Sanjit A. Seshia. A theory of mutations with applications to vacuity, coverage, and fault tolerance. In *FMCAD*, pages 1–9. IEEE, 2008.

Marta Z. Kwiatkowska, Alexandru Mereacre, Nicola Paoletti, and Andrea Patane. Synthesising robust and optimal parameters for cardiac pacemakers using symbolic and evolutionary computation techniques. In *HSB*, volume 9271 of *LNCS*, pages 119–140. Springer, 2015.

Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *AAAI*, pages 1536–1543. AAAI Press, 2019.

Kim Guldstrand Larsen and Wang Yi. Time-abstracted bisimulation: Implicit specifications and decidability. *Inf. Comput.*, 134(2):75–101, 1997.

Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *RTSS*, pages 298–307. IEEE Computer Society, 1995.

Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *CPAIOR*, volume 7874 of *LNCS*, pages 160–175. Springer, 2013. ISBN 978-3-642-38170-6.

Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *JAR*, 40(1):1–33, 2008.

Mark H. Liffiton, Maher N. Mneimneh, Inês Lynce, Zaher S. Andraus, João Marques-Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints An Int. J.*, 14(4):415–442, 2009.

Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21 (2):223–250, 2016.

Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In *TACAS*, volume 5505 of *LNCS*, pages 54–57. Springer, 2009.

Shaofan Liu and Jie Luo. FMUS2: An efficient algorithm to compute minimal unsatisfiable subsets. In *AISC*, volume 11110 of *LNCS*, pages 104–118. Springer, 2018.

Florian Lonsing and Uwe Egly. Qratpre+: Effective QBF preprocessing via strong redundancy properties. In *SAT*, volume 11628 of *LNCS*, pages 203–210. Springer, 2019.

Jie Luo and Shaofan Liu. Accelerating MUS enumeration by inconsistency graph partitioning. *Science China Information Sciences*, 62 (11):212104, 2019.

João Marques-Silva and Mikoláš Janota. On the query complexity of selecting few minimal sets. *Electronic Colloquium on Computational Complexity (ECCC)*, 21:31, 2014.

João Marques-Silva and Inês Lynce. On improving MUS extraction algorithms. In *SAT*, volume 6695 of *LNCS*, pages 159–173. Springer, 2011.

João Marques-Silva, Federico Heras, Mikoláš Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622. IJCAI/AAAI, 2013a.

João Marques-Silva, Mikoláš Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, volume 8044 of *LNCS*, pages 592–607. Springer, 2013b.

João Marques-Silva, Alexey Ignatiev, António Morgado, Vasco M. Manquinho, and Inês Lynce. Efficient autarkies. In *ECAI*, volume 263 of *FAIA*, pages 603–608. IOS Press, 2014.

João Marques-Silva, Mikoláš Janota, and Carlos Mencía. Minimal sets on propositional formulae. Problems and reductions. *Artif. Intell.*, 252:22–50, 2017.

Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, volume 2619 of *LNCS*, pages 2–17. Springer, 2003.

Alain Mebsout and Cesare Tinelli. Proof certificates for SMT-based model checkers for infinite-state systems. In *FMCAD*, pages 117–124. IEEE, 2016.

Carlos Mencía and João Marques-Silva. Reasoning about strong inconsistency in ASP. In *SAT*, volume 12178 of *LNCS*, pages 332–342. Springer, 2020.

Carlos Mencía, Alessandro Previti, and João Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979. AAAI Press, 2015.

Carlos Mencía, Alexey Ignatiev, Alessandro Previti, and João Marques-Silva. MCS extraction with sublinear oracle queries. In *SAT*, volume 9710 of *LNCS*, pages 342–360. Springer, 2016.

Carlos Mencía, Oliver Kullmann, Alexey Ignatiev, and João Marques-Silva. On computing the union of MUSes. In *SAT*, volume 11628 of *LNCS*, pages 211–221. Springer, 2019.

Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT (FOCS)*, pages 125–129. IEEE Computer Society, 1972.

Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

Sibylle Möhle and Armin Biere. Dualizing projected model counting. In *ICTAI*, pages 702–709. IEEE, 2018.

Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.

Kedian Mu. Formulas free from inconsistency: An atom-centric characterization in priest's minimally inconsistent LP. *J. Artif. Intell. Res.*, 66:279–296, 2019.

Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpsat. In *Canadian Conference on AI*, volume 7310 of *LNCS*, pages 356–361. Springer, 2012.

Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam, and Mats Per Erik Heimdahl. Compositional verification of a medical device system. In *HILT*, pages 51–64. ACM, 2013.

Anitha Murugesan, Michael W. Whalen, Elaheh Ghassabani, and Mats Per Erik Heimdahl. Complete traceability for requirements in satisfaction arguments. In *RE*, pages 359–364. IEEE Computer Society, 2016.

Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *FMCAD*, pages 221–229. IEEE, 2010.

Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *JSAT*, 9: 27–51, 2014.

Nina Narodytska, Nikolaj Bjørner, Maria-Cristina Marinescu, and Mooly Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361. ijcai.org, 2018.

Radek Pelánek. BEEM: benchmarks for explicit model checkers. In *SPIN*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

Marek Piotrów. UWrMaxSat - a new MiniSat+-based solver in MaxSAT Evaluation 2019. *MaxSAT Evaluation 2019*, page 11, 2019.

Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *AAAI*. AAAI Press, 2013. ISBN 978-1-57735-615-8.

Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Improving MCS enumeration via caching. In *SAT*, volume 10491 of *LNCS*, pages 184–194. Springer, 2017.

Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *AAAI*, pages 6633–6640. AAAI Press, 2018.

Markus N. Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In *FMCAD*, pages 136–143. IEEE, 2015.

Markus N. Rabe, Leander Tentrup, Cameron Rasmussen, and Sanjit A. Seshia. Understanding and extending incremental determinization for 2QBF. In *CAV (2)*, volume 10982 of *LNCS*, pages 256–274. Springer, 2018.

Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT*, 2004.

Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In *AAAI*, pages 475–482. AAAI Press / The MIT Press, 2005.

Palash Sashittal and Mohammed El-Kebir. Sampling and summarizing transmission trees with multi-strain infections. *Bioinformatics*, 36(Supplement_1):i362–i370, 2020.

Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In *IJCAI*, pages 1169–1176. ijcai.org, 2019.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

Mate Soos and Kuldeep S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *AAAI*, pages 1592–1599. AAAI Press, 2019.

Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, volume 5584 of *LNCS*, pages 244–257. Springer, 2009.

Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *CAV (1)*, volume 12224 of *LNCS*, pages 463–484. Springer, 2020.

Emanuel Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

Roni Tzvi Stern, Meir Kalech, Alexander Feldman, and Gregory M. Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*. AAAI Press, 2012.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In *Haskell*, pages 72–83. ACM, 2003.

Matthias Thimm. On the evaluation of inconsistency measures. *Measuring Inconsistency in Information*, 73, 2018.

Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In *SAT*, volume 4121 of *LNCS*, pages 424–429. Springer, 2006.

Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

Farn Wang. Formal verification of timed systems: a survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, 2004.

Michael W. Whalen, Darren D. Cofer, Steven P. Miller, Bruce H. Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In *FMICS*, volume 4916 of *LNCS*, pages 68–84. Springer, 2007.

Michael W. Whalen, Gregory Gay, Dongjiang You, Mats Per Erik Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *ICSE*, pages 102–111. IEEE Computer Society, 2013.

Siert Wieringa. Understanding, improving and parallelizing MUS finding using model rotation. In *CP*, volume 7514 of *LNCS*, pages 672–687. Springer, 2012.

Dongjiang You, Sanjai Rayadurgam, Michael W. Whalen, Mats Per Erik Heimdahl, and Gregory Gay. Efficient observability-based test generation by dynamic symbolic execution. In *ISSRE*, pages 228–238. IEEE Computer Society, 2015.

Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. *SAT*, 3, 2003.