CARL VON OSSIETZKY universität OLDENBURG

BERICHTE

AUS DEM DEPARTMENT FÜR INFORMATIK

der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften

Herausgeber: Die Professorinnen und Professoren des Departments für Informatik

An Algorithmic Framework for Checking Coverability in Well-Structured Transition Systems

Tim Strazny

Dissertation

Nummer 01/14 – Januar 2014

ISSN 1867-9218

CARL VON OSSIETZKY universität OLDENBURG

BERICHTE

AUS DEM DEPARTMENT FÜR INFORMATIK
der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften

Herausgeber: Die Professorinnen und Professoren
des Departments für Informatik

An Algorithmic Framework for Checking Coverability in Well-Structured Transition Systems

Tim Strazny

Dissertation

Nummer 01/14 – Januar 2014

ISSN 1867-9218

Gutachter:	Prof. Dr. E.-R. Olderog
	Prof. Dr. A. Podelski (Uni Freiburg)

Datum der Einreichung: 15.08.2013
Datum der Verteidigung: 16.12.2013

© 2014 by the author

Author's address:
Tim Strazny
Fakultät II, Department für Informatik
Abteilung „Entwicklung korrekter Systeme"
26111 Oldenburg
Germany

E-mail: Tim.Strazny@Informatik.Uni-Oldenburg.DE

CARL
VON
OSSIETZKY
universität OLDENBURG
FAKULTÄT II
INFORMATIK, WIRTSCHAFTS-
UND RECHTSWISSENSCHAFTEN

Carl von Ossietzky Unviersität Oldenburg
Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

An Algorithmic Framework for Checking Coverability in Well-Structured Transition Systems

Dissertation zur Erlangung des Grades eines Doktors der Naturwisschenschaften

vorgelegt von

Dipl.-Inform. Tim Strazny

Oldenburg, 15. August 2013

Abstract

Well-structured transition systems are an abstract class of infinite-state systems with transitions that are compatible with a simulation relation. In the context of automatic verification of these models, it often suffices to consider certain *coverability* problems which can be decided by the basic backward analysis algorithm introduced by Abdulla et al. When instantiating this algorithm for concrete system classes like extended Petri nets and lossy channel systems, similar questions for suitable optimizations and data structures have to be answered repeatedly.
In this thesis we present an abstract algorithmic framework based on the basic backward analysis algorithm. We introduce functions for witness traces and generalized predecessors that encompass well-known optimizations such as pruning and partial-order reduction as instances. With backward acceleration, a novel optimization is presented that is independent of the concrete system class. Moreover, we discuss search strategies inspired by the A^* algorithm and offer a general approach for the implementation of practical data structures. An empirical performance evaluation of the newly created reference implementation of the framework shows excellent results.

Zusammenfassung

Eine abstrakte Klasse zustandsunendlicher Systeme sind die sogenannten wohlstrukturierten Transitionssysteme, deren Transitionen mit einer Simulationsrelation harmonieren. Bei der automatischen Verifikation dieser Modelle genügt es häufig bestimmte Probleme der *Überdeckbarkeit* zu betrachten, die durch den Rückwärtsalgorithmus von Abdulla et al. entschieden werden. Bei der Instanziierung des Algorithmus für konkrete Systemklassen wie erweiterte Petrinetze und Lossy Channel Systems sind wiederholt Fragen nach geeigneten Optimierungen und Datenstrukturen zu beantworten.
In dieser Arbeit stellen wir ein auf dem Rückwärtsalgorithmus basierendes abstraktes Rahmenwerk vor. Dabei werden Funktionen für Zeugenpfade und verallgemeinerte Vorgänger eingeführt, die es erlauben, bekannte Optimierungsverfahren wie Pruning und Partial-Order Reduction als Instanzen zu verstehen. Mit Backward Acceleration wird eine neue Optimierung vorgestellt, die unabhängig von der konkreten Systemklasse ist. Ferner besprechen wir vom A^*-Algorithmus inspirierte Suchstrategien und bieten einen allgemeinen Ansatz zur Implementierung geeigneter Datenstrukturen. Eine empirische Laufzeitauswertung des entstandenen Programms zeigt ausgezeichnete Ergebnisse.

Acknowledgements

Throughout the undertaking of writing this thesis, hardly ever was I under the impression of being alone. There were moments of shared joy over finding new and powerful formal tools and there were moments in which I felt like a proverbial dwarf standing on the shoulders of giants. Writing this thesis was tough at times but with your help I did it.

First and foremost I have to thank Ernst-Rüdiger Olderog for being such an enjoyable, kind, and benevolent character. I consider myself lucky to have had you as my supervisor. Roland Meyer, thank you for sparking my interest in the topic of this thesis, providing help and advice, and leading by example. Thank you for being a friend. Sven Linker, thank you for the many discussions, double-checking my proofs, and most importantly, for being a friend way beyond the boundaries of academia. The three of you kept my academic wheel spinning.
A big thank you goes to the board of examiners, for investing their precious time and giving positive feedback: my supervisor Ernst-Rüdiger Olderog, the external examiner Andreas Podelski, chairwoman Annegret Habel, and Elke Wilkeit.

I am quite fond of the research and teaching environment arching over three working groups, the many talks at lunch, and the walks in the Haarenniederung. Thanks for providing and (formerly) being a part of that environment to (in alphabetical order) Annegret Habel, André Platzer, Björn Engelmann, Eike Best, Elke Wilkeit, Ernst-Rüdiger Olderog,

Hans Fleischhack, Hendrik Radke, Ingo Brückner, Jan-Daid Quesel, Johannes Faber, Mani Swaminathan, Martin Hilscher, Nils Erik Flick, Roland Meyer, Sibylle Fröschle, Stephanie Kemper, Sören Jeserich, and Sven Linker.

For additional input, support with their tools, and fruitful discussions I have to thank Jochen Hoenicke, Eike Möhlmann, Alexander Kaiser, Pierre Ganty, and Phillipe Schnoebelen.
To the proof readers: Ernst-Rüdiger Olderog, Sven Linker, Jan-David Quesel, Jörn Syrbe, Martin Hilscher, Maike Schwammberger, and Manuel Gieseking. Thanks for your valuable feedback and the many hours that sometimes unwelcome task took out of your schedule.

Where would I be without the neverending support of my family, Frauke Butz-Strazny, Klaus Strazny, and Heide Röder. When I was in need you were there. You had the right words, the right questions and an endless supply of affirmation. It is so much easier with you by my side, so much more worthwhile. Thank you.

Contents

I An Algorithmic Framework 1

1 Introduction 3
1.1 Contribution 5
1.2 Related Approaches 6
1.3 Thesis Structure 7
1.4 Sources 8
1.5 How to Read this Thesis 8

2 Preliminaries 11
2.1 First Glance: Petri Nets 12
2.2 Basics of Well-Structure 16
2.3 Well-Structured Transition Systems 19
2.4 The Coverability Problem 20
2.5 Decidability Results 30
2.6 Well-Structured Labelled Transition Systems 34
2.7 Instances of WSLTSs 36
2.7.1 Petri Nets 36
2.7.2 Petri Nets With Transfer 37
2.7.3 Lossy Channel Systems 40
2.8 Program Correctness 43

3 Algorithmic Framework **51**
3.1 Proof of the Basic BR 52
3.1.1 Partial Correctness 53
3.1.2 Termination 63
3.1.3 Total Correctness 68
3.2 Extending the Basic BR 69
3.3 Framework 79
3.4 Search Space Constructions 81
3.4.1 Optimized Predecessors and Distance Reduction . 82
3.4.2 Witness Traces and Search Strategies 84
3.5 Arguments for Termination and Correctness 85

4 Proof of the Algorithmic Framework **87**
4.1 Preliminaries 88
4.2 Partial Correctness 92
4.3 Termination 110
4.4 Total Correctness 124
4.5 Differences in the Partial Correctness Proofs 125

II Instantiating the Framework **127**

5 SSC Instances and Guided Search **129**
5.1 Backward Acceleration – A Novel Approach 131
5.1.1 Backward Acceleration is an SSC 136
5.1.2 Practical Approach to Backward Acceleration . . . 139
5.1.3 Computation of Maximal Extensions for Petri Nets 141
5.2 Pruning 143
5.3 Partial-Order Reduction 147
5.4 Combination of Search Space Constructions 152
5.5 Guided Search 155
5.5.1 Syntactic Distance and Syntactic Weight 156

6 Data Structures **159**
6.1 Related Work 160
6.2 Operations 161
6.3 From Necessary Conditions to Equivalence Classes 164
6.4 Equality Conditions 167

6.5 Total Order Conditions 168
6.6 Subset Conditions 170
6.6.1 Full Powerset Search Trees 172
6.6.2 Relaxed Powerset Search Trees 176
6.6.3 Preliminary Experiments 184
6.7 Hierarchy of Necessary Conditions 185
6.7.1 Preliminary Experiments 187
6.8 Select Operation 189

7 Reference Implementation and Experiments 191
7.1 Reference Implementation BW 192
7.1.1 Architecture 193
7.1.2 Extensibility 196
7.1.3 Unit Tests 204
7.1.4 Usage: An Example 206
7.2 Experiments 207
7.2.1 Lossy Channel Systems 208
7.2.2 PN and PNT Benchmark Models 208
7.2.3 Experiments on SSCs and Search Guidance 211
7.2.4 Tool Comparison 214

8 Conclusion 225
8.1 Summary 225
8.2 Directions for Future Work 227
8.2.1 Algorithmic Framework and Distance 227
8.2.2 Implementation 229
8.2.3 Well-Structured Transition Systems 230

III Appendices 233

A Implementation Details 235
A.1 Generics 235
A.2 Command-Line Options 236

B Sources of Selected Case Studies 239
B.1 Holonic Transportation System 239
B.2 PNCSA Protocol 246

B.3 Kanban Production System 250
B.4 Delegate Buffer Program 255

C Benchmark Data **265**
C.1 Data Structures 266
C.2 Search Space Constructions and Search Guidance 269
C.3 Tool Comparison 286

Bibliography **293**

Index **313**

List of Figures

2.1 Example of a Petri net and different markings. 12
2.2 P/T Petri net N_{ex} 14
2.3 Coverability graph of N_{ex} 16
2.4 Upward-compatibility of $\rightarrow$ for WSTSs. 20
2.5 Witness trace for $m_\alpha \hookrightarrow m_\Omega$ in N_{ex} 23
2.6 Upward-compatibility in the proof of Lemma 2.8 on p. 26. 26
2.7 Backward exploration of N_{ex} , starting in m_Ω 32
2.8 Backward exploration of N_{ex} , starting in $p_2 + p_4$ 33
2.9 Upward-compatibility of $\rightarrow$ for WSLTSs. 35
2.10 P/T Petri net example: Mutual exclusion of p_2 and p_4 . . 37
2.11 Petri net with transfer example: Buffered Producer / Consumer. 39
2.12 Lossy channel system example: Alternating bit protocol. . 42
2.13 Example proof outline for total correctness: FACTORIAL. . 49

3.1 Upward-compatibility in the proof of Lemma 3.1 on p. 53. 54
3.2 Proof outline for partial correctness of Alg. 2.1 on p. 30. . 62
3.3 Proof outline for termination of Alg. 2.1 on p. 30. 67

4.1 Proof outline for partial correctness of Alg. 3.7 on p. 80. . 95
4.2 Proof outline for partial correctness – INIT. 97
4.3 Proof outline for partial correctness – NEXT_STATE. . . . 98

4.4 Proof outline for partial correctness – FOUND__TRACE. . . 100
4.5 Proof outline for partial correctness – ADD__PREDECESSORS. 105
4.6 Proof outline for partial correctness – PROCESS__NEW__-STATE. 107
4.7 Proof outline for partial correctness – UPDATE__TRACE. . 109
4.8 Proof outline for termination of Alg. 3.7 on p. 80. 113
4.9 Proof outline for termination – INIT. 114
4.10 Proof outline for termination – NEXT__STATE. 116
4.11 Proof outline for termination – FOUND__TRACE. 117
4.12 Proof outline for termination – ADD__PREDECESSORS. . . 120
4.13 Proof outline for termination – PROCESS__NEW__STATE. . 121
4.14 Proof outline for termination – UPDATE__TRACE. 123

5.1 Schematic example search space. 130
5.2 Recurring patterns in the search space of the Kanban case study. 131
5.3 Schematic example search space for acceleration. 134
5.4 Exploration of N_{ex}, m_Ω with backward acceleration. . . . 135
5.5 Example of constructing a candidate sequence 140
5.6 Schematic example search space for pruning. 144
5.7 Exploration of N_{ex}, m_Ω with pruning. 145
5.8 Schematic example search space for POR. 147
5.9 Exploration of N_{ex}, m_Ω with partial-order reduction. . . . 149
5.10 Exploration of N_{ex}, m_Ω with combined SSC PORPA. . . . 154
5.11 Syntactic distance and weight in Petri net N_{ex} 158

6.1 Updating minimal basis X with x, x' 163
6.2 Example of a full powerset search tree. 174
6.3 Examples of relaxed powerset search trees. 179
6.4 Merging a tree extension with a PST. 182
6.5 Runtime benefit of partitioning via subset conditions. . . 186
6.6 Comparison of hierarchies of total order and subset conditions. 188

7.1 Dependencies between classes for data structures. 200
7.2 Comparison of effects of SSCs and search guidance. 213
7.3 Tool comparison: Median time. 217
7.4 Tool comparison: Maximum time. 218

7.5 Tool comparison: 97%-confidence interval. 218
7.6 Detailed benchmark results I. 221
7.7 Detailed benchmark results II. 222

B.1 Overview of a Petri net representation of the HTS 240
B.2 Considered part of the PNCSA protocol ([Fin93]) 247
B.3 A Kanban production cell according to [MBC+95] 251
B.4 A Kanban production cell similar to [CM97] 252
B.5 Considered Kanban production system ([CM97]) 253

List of Tables

7.1 Properties of benchmark case studies 209
7.2 Selected benchmark data for AGIP vs. GIP. 214

C.1 Benchmark data for Tree Height in Sect. 6.6.3 267
C.2 Benchmark data for f_{Σ} before f_{supp} in Sect. 6.7.1 268
C.3 Benchmark data for f_{supp} before f_{Σ} in Sect. 6.7.1 268
C.4 Benchmark data for AGIP in Sect. 7.2.3 270
C.5 Benchmark data for GIP in Sect. 7.2.3 271
C.6 Benchmark data for AGI in Sect. 7.2.3 272
C.7 Benchmark data for GI in Sect. 7.2.3 273
C.8 Benchmark data for AGP in Sect. 7.2.3 274
C.9 Benchmark data for GP in Sect. 7.2.3 275
C.10 Benchmark data for AG in Sect. 7.2.3 276
C.11 Benchmark data for G in Sect. 7.2.3 277
C.12 Benchmark data for AIP in Sect. 7.2.3 278
C.13 Benchmark data for IP in Sect. 7.2.3 279
C.14 Benchmark data for AI in Sect. 7.2.3 280
C.15 Benchmark data for I in Sect. 7.2.3 281
C.16 Benchmark data for AP in Sect. 7.2.3 282
C.17 Benchmark data for P in Sect. 7.2.3 283
C.18 Benchmark data for A in Sect. 7.2.3 284
C.19 Benchmark data for $\emptyset$ in Sect. 7.2.3 285

C.20 Benchmark data for Framework in Sect. 7.2.4 288
C.21 Benchmark data for Petruchio/BW in Sect. 7.2.4 . . . 289
C.22 Benchmark data for MIST2 in Sect. 7.2.4 290
C.23 Benchmark data for BFC 2.0 in Sect. 7.2.4 291
C.24 Benchmark data for BFC 1.0 in Sect. 7.2.4 292

List of Algorithms

2.1 Basic backward reachability analysis BASIC BR 30
2.2 Example algorithm for program correctness: FACTORIAL. . 49

3.1 BR Extension I: Minimization. 70
3.2 BR Extension II: Disjoint $\uparrow V$ and W 70
3.3 BR Extension III: Shortcut. 72
3.4 BR Extension IV: Explicit Inner Loop. 73
3.5 BR Extension V: Optimized Predecessors. 74
3.6 BR Extension VI: Witness Traces. 78
3.7 ALGORITHMIC FRAMEWORK $back(select, opb, wit)$ 80

4.1 Subalgorithms of the ALGORITHMIC FRAMEWORK. 89

7.1 Benchmarking SSCs and search guidance. 212

8.1 Suggested algorithm for a reduced memory footprint. . . . 228

List of Listings

7.1 Interfaces for states and transitions 194
7.2 Actual methods to implement for a state 196
7.3 Interface for a model 197
7.4 Actual methods to implement for a model 198
7.5 PN TokenSumAbstraction 199
7.6 PN SupportAbstraction 201
7.7 PN abstraction hierarchy 201
7.8 Parser interface 202
7.9 Example output of the reference implementation 206

A.1 Choice of generic types for states 236
A.2 Command-line options of the reference implementation . . 237

B.1 π -Calculus process of the holonic transportation system ([MKS09]) 241
B.2 Source of the PNCSA protocol 246
B.3 Source of the Kanban production system 254
B.4 Java source of class `BoundedBufferWithDelegates` ([Lea99]) 256
B.5 Source of the delegate buffer program 258

PART I

An Algorithmic Framework

CHAPTER ONE

Introduction

"…when things are simple, fewer mistakes are made. The most expensive part of a building is the mistakes."
— Ken Follett, *The Pillars of the Earth*

Contents

1.1 Contribution 5
1.2 Related Approaches 6
1.3 Thesis Structure 7
1.4 Sources 8
1.5 How to Read this Thesis 8

In our modern world, computers are ubiquitous. Unfortunately, so are costly programming errors that effect diverse fields: The crashing of the AT&T long distance telephone network in 1990 (which carried over 115 million calls daily) might be seen as annoying, whereas the 2004 bug in the German A2LL software which caused unemployment benefit to be sent to invalid bank accounts was a financial disaster for the unemployed. The software bug that lead to the 2003 North American blackout resulted in contamination of drinking water due to failing pumps and massive disruption of communication services and public transportation.

It left 55 million people without electrical energy for up to 48 hours. Sadly, there are examples where programming errors in systems resulted in direct physical harm—such as the accidents in context of the Therac-25 radiation therapy machine in the 1980s where patients were subjected to massive overdoses of radiation. The examples indicate that software errors can influence diverse areas of life and that they occur on a multitude of platforms. The most common method to find bugs is testing: trying out a program with different input values and observing whether unwanted behaviour commences. This technique is of particular importance in the context of large software systems. However, while tests can show the existence of bugs, tests do not prove their absence.

"How can one check a routine in the sense of making sure that it is right?" is the introductory question to Turing's 1949 paper *Checking a Large Routine* [Tur49][1]. He suggests to annotate programs with "assertions which can be checked individually, and from which the correctness of the whole programme easily follows."

The field of theoretical computer science has since evolved and (amongst others) lifted Turing's idea of manual axiomatic correctness proofs for programs to the *computer aided verification* of models of systems. While the automation of the verification process is ruled impossible due to the undecidability of non-trivial properties of general models, restricting the considered models allows for an automated analysis.

In this work we focus on *model checking* of *well-structured transition systems* where an automatic process determines if a model with few restrictions can evolve to reach a given set of states. From our perspective, the beauty of well-structured transition systems (WSTSs) is their ubiquity, or, as Finkel and Schnoebelen put it in the title of their 2001 paper [FS01], *Well-Structured Transition Systems Everywhere!*

Well-structured transition systems are a framework for the automatic verification of infinite-state systems that was found independently by Finkel [Fin87] and Abdulla [AČJT96] when they worked on generalizations of decision procedures that were known for particular models. Technically, WSTSs are infinite-state transition systems where the transition relation is monotonic w.r.t. a well-quasi ordering on the states. Monotonicity means larger states can imitate the behaviour of smaller states

[1] The original publication contained several transcription errors. A corrected and commented version was published in 1984 [MJ84].

or, phrased differently, smaller states let us conclude about the behaviour of larger ones. The well-quasi ordering guarantees finite representations of infinite sets of states, forming the foundation of termination results for verification algorithms.

Indeed, one goal of WSTS research is the generalization of decision procedures that are known for particular models. Most notably, in [AČJT96] Abdulla et al. extended the decidability result for coverability in Petri nets [KM69] to a decision procedure for coverability that works for general WSTSs. Given some target state, coverability asks for a path to a state that dominates the target. For many systems, coverability is what is needed for (safety) verification. To give an example, mutual exclusion immediately relates to a coverability query. In contrast to the Karp & Miller algorithm for Petri nets, the extension to WSTSs works backwards and maintains a set of minimal elements rather than limit configurations. It is this backward algorithm that we generalize to a framework for rapid prototyping programs for model checking WSTSs.

1.1. Contribution

We conduct an *axiomatic proof* of the basic backward analysis procedure for well-structured transition systems.

We develop an *algorithmic framework*—an extension of the basic analysis—for checking coverability in well-structured transition systems and introduce an abstract *distance* function to formulate constraints for adequate instantiations of the framework via so-called search space constructions. We show the *total correctness* of the framework via an axiomatic Hoare-style proof.

We present a novel search space construction called *backward acceleration* which cuts recurring paths from the analysis. We show that our framework is a *conservative generalization* of the backward analysis in the sense that the established optimization techniques of pruning and partial-order reduction are search space constructions. We present *search strategies* for specific classes of well-structured transition systems to guide the analysis.

We develop a general approach to induce data structures via *necessary conditions*. These data structures represent infinite sets of states and can be used in the analysis. We identify three *prototypical conditions* and discuss the use of well-understood data structures for checking two

types of conditions. Subsequently, we introduce the *powerset search tree* data structure for checking conditions of the remaining type and conduct preliminary *experiments* to test the effectiveness.

We present a *reference implementation* of the framework and the methods developed in this thesis. For this software framework we lay focus on *extensibility*, meaning that we enable the user to easily construct *prototype* coverability checkers for user-defined system classes by providing new software modules with little effort. We discuss the results of a comprehensive *experimental evaluation* of the implementation and a *comparison* with state-of-the-art programs.

1.2. Related Approaches

Due to their ubiquity, well-structured transition systems and many of their concrete system classes have been studied extensively.

In 1978 Rackoff showed that the coverability problem for Petri nets has a lower-bound exponential space requirement [Rac78] (the bound was refined by [RY85]). Recently, Bozzelli and Ganty showed that the basic backward coverability analysis [AČJT96] is optimal in the Petri net case [BG11].

In addition to the backward analysis, several other efforts have been carried out, mainly for the class of (extended) Petri nets. Abdulla et al. presented a SAT-based approach for Petri net coverability where they employ unfolding techniques for unbounded nets [AIN04]. For Petri nets, a minimal coverability set can be constructed that allows for deciding coverability problems [FRSV03, GRV07, VH12]. Leuschel and Lehmann examined the use of partial deduction to attack coverability problems for Petri nets [LL00]. Ganty investigated an automatic abstraction refinement procedure for coverability in Petri nets [GRV08, Gan07].

In 2005, Bingham and Hu presented a backward analysis for a subclass of WSTSs based on the data structure of binary decision diagrams [Bin05, BH05] and Geeraerts et al. introduced the first forward analysis for the whole class of WSTSs called *expand, enlarge, and check* (EEC) which constructs a converging sequence of over- and under-approximations of the system [GRV06b, GRV05]. Shortly after the introduction of the new EEC approach, Ganty et al. presented an abstract interpretation based

approach to solve coverability for WSTSs using a forward algorithm [GRV06a].
In 2008, Dimitrova and Podelski showed that the *lazy abstraction* of Henzinger et al. [HJMS02] (a partly automated counter-example guided abstraction refinement technique [CGJ+00]) can be effectively instantiated with deterministic control for WSTSs and that it is a decision procedure for the coverability problem [DP08].
Most recently, Kaiser et al. implemented a novel approach of target set widening for coverability analysis, employing a combined forward and backward search for WSTSs [KKW12]. Furthermore, Kloos et al. present an incremental, inductive (IC3) procedure to check coverability for a large subclass of WSTSs [KMNP13].
Even years after the first general decidability result of Abdulla et al, the interest in the automatic analysis of coverability problems of well-structured transition systems remains unbroken. While there exists a plethora of approaches to solve coverability problems for (extended) Petri nets and several well-received methods for (subclasses of) WSTSs, we are not aware of any simple algorithmic framework that has the stated goal to be easily instantiated for new classes of WSTSs with little effort.

1.3. Thesis Structure

In the first part of this thesis we establish an abstract algorithmic framework for checking coverability in well-structured transition systems. Basic definitions and results are captured in Chapter 2. There, we also recapitulate some well-known instances of well-structured transition systems and introduce a running example. In Chapter 3 we present a novel Hoare-style proof of the basic backward analysis and develop our algorithmic framework by extending the basic algorithm. The detailed proof of our framework in Chapter 4 shows its (total) correctness.
The second part of the thesis is concerned with an intermediate concretization of our algorithmic framework that reduces the costs to instantiate it for specific models. Chapter 5 introduces a novel optimization for backward analysis of general well-structured transition systems and shows that established optimizations are instances of our framework. In that chapter, we also discuss the effect of strategies to guide the search during a coverability analysis. Chapter 6 shows how necessary conditions

for a well-quasi ordering are used to speed up the analysis and with powerset search trees a novel data structure is introduced. We present an extensible reference implementation of our framework in Chapter 7 and show its effectiveness and performance in an extensive comparison with several state-of-the-art tools. In Chapter 8 we summarize our work and sketch directions for future work.
The third part of the thesis consists of three appendices. In Appendix A we give some details on the reference implementation. Appendix B describes four selected case studies in detail. In Appendix C we list the numerical data of the experimental evaluation.

1.4. Sources

The roots of this thesis go back to the development of the PETRUCHIO tool [MS10, MKS09, Str07] which provides a verification environment for π-calculus processes. PETRUCHIO calculates the Petri net semantics [Mey09]. In the course of computing the semantics, a set of coverability problems have to be solved.
Motivated by this work, the present thesis develops a general algorithmic framework for solving coverability problems in well-structured transition systems. Parts of this thesis stem from joint work with Roland Meyer or have been partially published before. The backward acceleration we present in Chapter 5 was first introduced in [Str11]. Our algorithmic framework that we describe in this thesis was inaugurated in [SM12] and the results of this publication are used throughout this work.

1.5. How to Read this Thesis

While this thesis is intended to be read from front to back, the dependencies between the contents of this thesis allow for different selections of topics.
Unless already familiar with the notions of well-structured transition systems and coverability, we suggest reading Sections 2.2 to 2.6 of the preliminaries. To follow the examples, we advise to also read Section 2.1.
For readers who are interested in the Hoare-style proofs of the basic backward reachability analysis in Section 3.1 and the proof of our algorithmic

framework in Chapter 4, reading Section 2.8 on program correctness is mandatory.
Readers who want to concentrate on optimization techniques could read Chapter 5 and then turn to the experimental evaluation in Section 7.2.3.
Those readers who are concerned with implementation details should read Section 3.3 and Section 3.4 which introduce our algorithmic framework, Chapter 6 on data structures, and Section 7.1 where we describe our reference implementation. Reading Chapter 5 on optimizations is encouraged.

CHAPTER TWO

Preliminaries

Clutch it like a cornerstone. Otherwise it all comes down.
— Tool, *The Grudge*

Contents

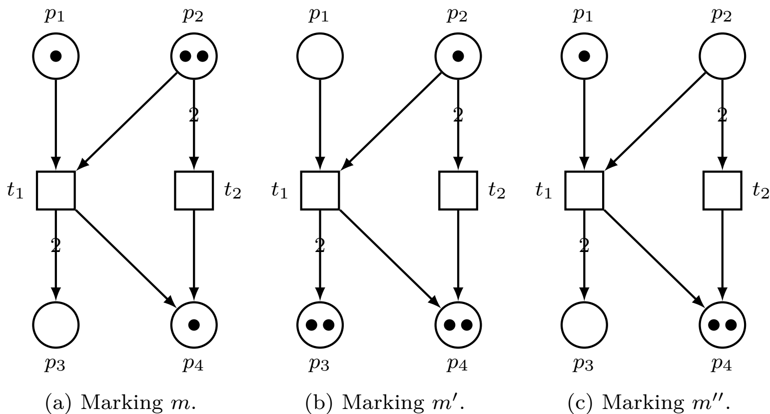
2.1 First Glance: Petri Nets 12
2.2 Basics of Well-Structure 16
2.3 Well-Structured Transition Systems 19
2.4 The Coverability Problem 20
2.5 Decidability Results 30
2.6 Well-Structured Labelled Transition Systems 34
2.7 Instances of WSLTSs 36
2.7.1 Petri Nets 36
2.7.2 Petri Nets With Transfer 37
2.7.3 Lossy Channel Systems 40
2.8 Program Correctness 43

To start with, we introduce the system class of Petri nets to provide tangible illustrations for the abstract concepts of the following sections.

With a running example at hand, we examine well-structured transition systems and the coverability problem. Following the discussion of decidability results and respective algorithms, we recall further instances of the general framework of well-structured transition systems: Petri nets with transfer, and lossy channel systems.

2.1. First Glance: Petri Nets

Figure 2.1 shows a simple Petri net together with three different states. The net consists of four *places* and two *transitions* that are connected via weighed *arcs*. A Petri net's state is defined by the distribution of *tokens* on its places. For example, in Fig. 2.1a there is one token on each of the places p_1 and p_4, two tokens on place p_2, and no token on place p_3. Before we discuss the semantics of Petri nets we turn to the formal definition of their syntax.



(a) Marking m. (b) Marking m'. (c) Marking m''.

Figure 2.1.: Example of a Petri net and different markings.

Definition 2.1 (Petri net). A *Petri net* (PN) is a triple $N = (P, T, W)$ where $P = \{p_1, \ldots, p_{|P|}\}$ is a finite set of *places*, T a finite set of *transitions*, and $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ is a *weight function*. Places and transitions are disjoint, $P \cap T = \emptyset$. (Cf. for example [PW08])

Graphically, places are represented by circles with dots for tokens, transitions are drawn as boxes. Net elements e_1, e_2 (places or transitions) for which the weight function yields a positive value, i.e. $W(e_1, e_2) > 0$, are connected via *arcs* (arrows) labelled with the weight (labels of weight 1 are omitted).

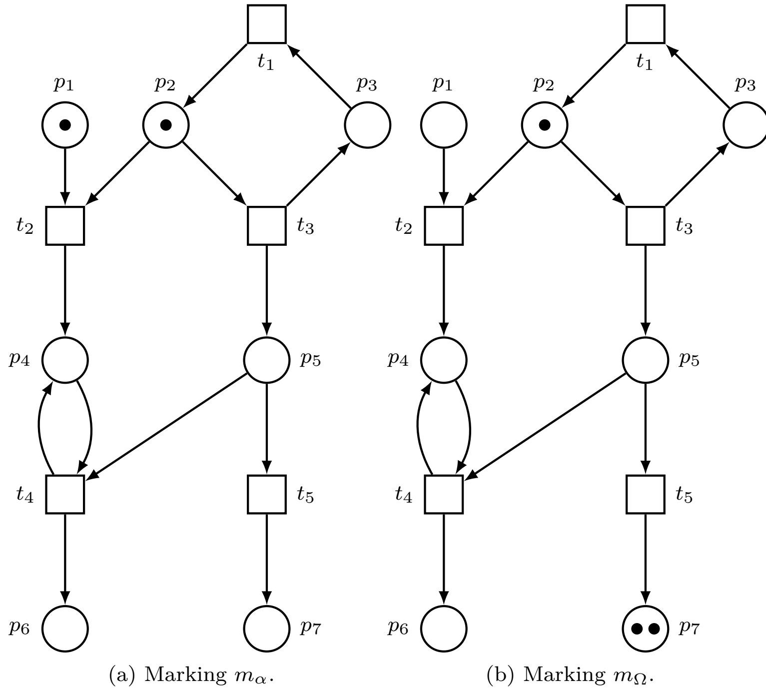
For any net element e (place or transition), its preset $^\bullet e$ consists of all net elements f from which arcs with non-zero weights originate and point towards e, i.e. $^\bullet e = \{ f \in P \cup T \mid W(f, e) \neq 0 \}$. Analogously, the net element's postset $e^\bullet$ consists of all net elements f that are connected to e by arcs with non-zero weights that originate in e, i.e. $e^\bullet = \{ f \in P \cup T \mid W(e, f) \neq 0 \}$.

The semantics of Petri nets relies on markings that can be understood as the state of a Petri net at runtime. Formally, a *marking* is a vector $m \in \mathbb{N}^P$ that assigns a natural number to every place. The execution of a transition t, called *firing* and denoted by $m_1 \xrightarrow{t} m_2$, changes the token count. But as markings are defined to be semi-positive, there is a restriction. A transition can only be fired if the places in its preset contain enough tokens. Formally, transition $t \in T$ is *enabled in m* if $m \geq W(-, t)$, where $W(-, t)$ is the vector $(W(p_1, t), \ldots, W(p_{|P|}, t))^\mathsf{T}$. The ordering among vectors is defined component-wise. If the transition is enabled, its firing produces $W(t, p)$ tokens on every place p in its postset and, at the same time, consumes $W(p, t)$ tokens from the places in its preset: $m_1 \xrightarrow{t} m_2$ if t is enabled in m_1 and $m_2 = m_1 - W(-, t) + W(t, -)$. ◊

In marking m, depicted in Fig. 2.1a, both transitions t_1 and t_2 are enabled. Firing t_1 in m consumes one token each from p_1, p_2 and produces two tokens in p_3 and one in p_4. It results in marking m' shown in Fig. 2.1b, thus $m \xrightarrow{t_1} m'$. Firing t_2 in m leads to m'', represented in Fig. 2.1c. In both m' and m'' no transition is enabled.

We call these nets Place/Transition Petri nets or *P/T Petri nets* for short and discuss the differences to extended Petri nets in Sect. 2.7.2.

In Fig. 2.2, Petri net N_{ex}, our running example for the next chapters, is shown together with two different markings. The Petri net consists of five transitions $t_1, \ldots, t_5$ and seven places $p_1, \ldots, p_7$ of which exactly p_1 and p_2 are initially marked with one token each. A concise shorthand notation is $m_\alpha = p_1 + p_2$, which relates to so-called *multisets*. By m_α we denote the initial marking of Petri net N_{ex}. The marking is depicted in



(a) Marking m_α.

(b) Marking m_Ω.

Figure 2.2.: P/T Petri net N_{ex}.

Fig. 2.2a. Moreover, we introduce the final marking m_Ω of N_{ex}, shown in Fig. 2.2b, where p_2 contains one token and p_7 contains two tokens while all other places are empty. In the shorthand notation $m_\Omega = p_2 + 2p_7$.
The coverability problem for Petri nets asks whether a marking can be reached that *covers* the final marking, i.e, in the case of N_{ex}, "is '$m_\Omega + X$' reachable?" Where "$+X$" means to "add tokens to your liking."
Is it possible to reach a marking covering m_Ω when starting from m_α? Yes, via transition sequence $t_3\, t_1\, t_5\, t_3\, t_1\, t_5 = (t_3\, t_1\, t_5)^2$ for example. (If not stated differently, we order places lexicographically when writing out markings.) The marking reached is

$$m = (1, 1, 0, 0, 0, 0, 2)^\mathsf{T} = p_1 + p_2 + 2p_7$$

which covers m_Ω as it adds one token to p_1 in comparison to m_Ω. We write $m_\Omega \leq m$. The fact that the transition sequence $(t_3\, t_1)^2\, (t_5)^2$, which forms a simple reordering of the previous sequence, leads to the same marking is of interest in Sect. 5.3 on p. 147, where *partial-order reduction* is discussed.

Instead of guessing a transition sequence, let us take a more systematic approach that reveals certain properties of N_{ex}. The set of reachable markings together with transition firings forms a labelled transition system which is infinite if and only if the number of tokens on a place can grow arbitrarily. For any given P/T Petri net, one can construct the so-called *coverability graph* which overapproximates the net's labelled transition system by introducing *ω-markings*. These markings are vectors over $(\mathbb{N} \cup \{\, \omega \,\})^P$ and whenever a place contains ω tokens, there may be arbitrarily many tokens.

In Fig. 2.3, the coverability graph of N_{ex} is given and the two markings covering m_Ω are underlined. Each transition sequence of N_{ex} is contained within this graph and furthermore, due to the occurrence of the ω symbol in some markings, we know that places p_5, p_6, p_7 are *unbounded*, i.e. there are runs that can put arbitrary numbers of tokens on these places. However, the other places may never contain more than one token each. We will look into more of these details in Sect. 5.2 on p. 143 when we discuss *pruning*.

From the graph, we can see that many transition sequences lead to covering markings drawn bold. Please note that while every sequence of the net can be matched by a path in the graph, the converse does not hold as the Petri net may need to perform more repetitions of a subsequence than the sequence of the graph dictates. Nevertheless, for any ω-marking in a coverability graph and any natural number n, we can create a firing sequence of the net that puts at least n tokens in place of the ω's.

Unfortunately, it is very costly to construct a coverability graph as the number of ω-markings grows with non-primitive recursive complexity in the worst case. In practice, construction of complete coverability graphs is no viable option to check coverability problems for large Petri nets.

With Petri nets, we now have a simple visualization at hand and are ready for the details of well-structured transition systems and see how they apply to our running example N_{ex}. However, we return to Petri nets later in this chapter.

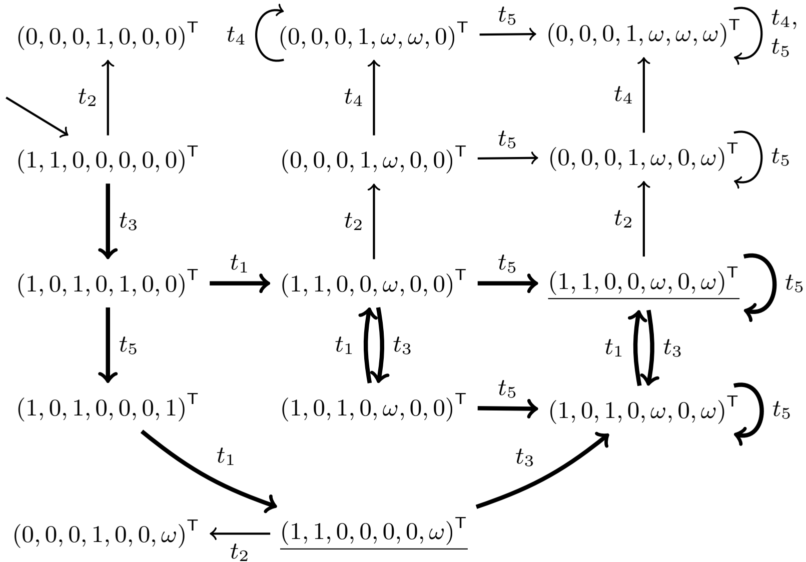


Figure 2.3.: Coverability graph of N_{ex}.

2.2. Basics of Well-Structure

A well-structured transition system consists of states, transitions and a well-quasi ordering. In order to investigate these systems algorithmically, we recall the definitions of (well-) quasi orderings and some of their properties.

Definition 2.2 (Quasi Ordering). A *quasi ordering* (QO) is a relation $\preceq$ over some set X s.t.

1. $\forall x \in X : x \preceq x$, i.e. $\preceq$ is reflexive, and
2. $\forall x, y, z \in X : (x \preceq y \wedge y \preceq z \Rightarrow x \preceq z)$, i.e. $\preceq$ is transitive.

By $x \prec y$ we abbreviate $x \preceq y \wedge y \not\preceq x$. ◊

Note that a QO is not antisymmetric, meaning that $x \preceq y \preceq x$ does *not* imply $x = y$. However, a QO $\preceq$ over X induces an equivalence relation

over X by $x \equiv_{\preceq} y$ iff $x \preceq y \wedge y \preceq x$ for $x, y \in X$. Thus, a QO induces a partial order over equivalence classes of $\equiv_{\preceq}$.
In extending quasi orderings by stipulating the *existence of comparable elements in infinite sequences*, well-quasi orderings are defined.

Definition 2.3 (Well-Quasi Ordering). A *well-quasi ordering* (WQO) is a quasi ordering $\preceq$ over some set X s.t. for any infinite sequence $x_0, x_1, \ldots \in X$, there exist indices $i < j$ with $x_i \preceq x_j$. ♢

Example 2.1. Remember the marking reached from m_α via transition sequence $(t_3\, t_1\, t_5)^2$ in our running example N_{ex} is

$$m = (1, 1, 0, 0, 0, 0, 2)^{\mathsf{T}}$$

which is component-wise greater or equal to $m_\Omega = (0, 1, 0, 0, 0, 0, 2)^{\mathsf{T}}$ and we wrote $m_\Omega \leq m$. The relation $\leq$ on markings is the well-quasi ordering we are interested in the PN setting. Further, Dickson's Lemma [Dic13] states that any component-wise partial order over vectors of natural numbers is a well-quasi ordering. In fact, it is even a well-partial order, i.e. it is antisymmetric: $x \leq y \leq x \Rightarrow x = y$. As an example, take vector $x = (0, 0, 0, 0, 5, 6, 7)^{\mathsf{T}}$. There are $5 \cdot 6 \cdot 7 = 210$ different vectors that are less than or equal to X. ♢

The first of the components that make up a well-structured transition system is fully defined. However, Higman [Hig52, Theorem 2.1] states the equivalence of the WQO property [Hig52, Theorem 2.1, (v)] to five more properties, four of which we recall over the course of this chapter in the lemmas 2.2 and 2.1 (well-foundedness and absence of infinite anti-chains)[1], 2.3 (infinite monotone subsequence), 2.6 (finite basis), and 2.9 (stabilization).[2] These properties are the building blocks of the decidability results for well-structured transition systems (cf. Sect. 2.5).
One of the properties of WQOs is that they are well-founded and contain no infinite anti-chains, which we capture in the definitions and the lemma that follow.

[1]The WQO property is implied by lemmas 2.2 (well-foundedness) and 2.1 (absence of infinite anti-chains) only in conjunction.

[2]Left out: "For a WQO $\preceq$ over X and $Y \subseteq X$, there exists a finite set basis(Y) s.t. basis(Y) $\subseteq Y \subseteq \uparrow$basis($Y$)" [Hig52, Theorem 2.1 (iii)], which simply states that any set Y has a finite basis (cf. Def. 2.7 and Lemma 2.6) w.r.t. the WQO that is contained in Y and that the basis' upward-closure contains Y. Higman cites [Bir40] for proofs of some implications and equivalences.

Definition 2.4 (Anti-Chain). Let $\preceq$ be a quasi-ordering over some set X. An *anti-chain* is a (possibly infinite) sequence $x_1, x_2, \ldots \in X$ of pairwise incomparable elements, i.e. for all indices i, j, elements x_i and x_j are incomparable: $x_i \npreceq x_j$ holds. ♢

Lemma 2.1 (Absence of Infinite Anti-Chains in Well-Quasi Orderings). A WQO $\preceq$ over some set X does not contain infinite anti-chains, i.e. there is no infinite sequence without comparable elements [Hig52, Theorem 2.1 (vi)].

Proof. Assume an infinite anti-chain, i.e. an infinite sequence $x_0, x_1, \ldots \in X$ of pairwise incomparable elements. This trivially contradicts the WQO property, as there exist indices $i < j$ of comparable elements s.t. $x_i \preceq x_j$. □

Definition 2.5 (Well-Foundedness). A relation $>$ over some set X is well-founded if no infinite strictly decreasing sequence exists, i.e. there is no infinite sequence $x_0 > x_1 > \cdots \in X$. ♢

Lemma 2.2 (Well-Foundedness of Well-Quasi Orderings). A well-quasi ordering $\preceq$ over some set X is well-founded, i.e. there is no infinite strictly decreasing sequence [Hig52, Theorem 2.1 (vi)].

Proof. Assume an infinite strictly decreasing sequence $x_0 \succ x_1 \succ \cdots \in X$. This trivially contradicts the WQO property, as there exist indices $i < j$ s.t. $x_i \preceq x_j$. □

The well-foundedness lemma denies the existence of infinite strictly decreasing sequences, whereas the following lemma lifts the WQO property from the existence of two comparable elements $x_i \preceq x_j$ to an infinite monotone subsequence.

Lemma 2.3 (Infinite Monotone Subsequence). For a WQO $\preceq$ over some set X, any infinite sequence $x_0, x_1, \ldots \in X$ contains an infinite monotone subsequence, i.e. there exist indices $i_0 < i_1 < \cdots$ s.t. $x_{i_0} \preceq x_{i_1} \preceq \cdots$. [Hig52, Theorem 2.1 (iv)]

Proof. Consider a WQO $\preceq$ over X and an infinite sequence $x_0, x_1, \ldots \in X$. Construct set $M = \{\, i \in \mathbb{N} \mid \forall j > i : x_i \npreceq x_j \,\}$. M has to be finite, otherwise it would lead to an infinite sequence $x_{i_0}, x_{i_1}, \ldots$ which would contradict the WQO property. Since M is finite, for any $i > \max(M)$

there is a $j > i$ s.t. $x_i \preceq x_j$. Thus, any $i_0 > \max(M)$ can start an infinite monotone subsequence $x_{i_0} \preceq x_{i_1} \preceq \cdots$. □

Equipped with the basic understanding of well-quasi orderings, well-structured transition systems follow naturally.

2.3. Well-Structured Transition Systems

Well-structured transition systems are (typically infinite) transition systems where the set of states is equipped with a well-quasi ordering.

Definition 2.6 (Well-Structured Transition System). A triplet $(S, \rightarrow, \preceq)$ is a *well-structured transition system* (WSTS) if S is a set of states, $\rightarrow \ \subseteq S \times S$ is a transition relation and $\preceq$ is a decidable[3] WQO. The transition relation is *upward-compatible* with $\preceq$, i.e.

$$\forall s_1, s_2, t_1 \in S : (s_1 \preceq t_1 \wedge s_1 \rightarrow s_2 \Rightarrow \exists t_2 \in S : s_2 \preceq t_2 \wedge t_1 \rightarrow^* t_2) . \quad \Diamond$$

Note that $\rightarrow^*$ is the reflexive, transitive closure of the transition relation.

Example 2.2. Upward-compatibility for Petri nets comes for free: If a transition is enabled in some marking m_1, we cannot disable that transition by adding more tokens to m. Even more so, if m_2 is reached by firing t from m_1, i.e. $m_1 \xrightarrow{t} m_2$, and m_1' covers m_1, i.e. we can represent it as $m_1' = m_1 + m$ for some marking m, then $m_1' \xrightarrow{t} m_2 + m$, where the summation of markings is applied component-wise. More formally:

$$\forall t \in T \ \forall m_1, m_2, m \in \mathbb{N}^P : (m_1 \xrightarrow{t} m_2 \Rightarrow m_1 + m \xrightarrow{t} m_2 + m) . \quad \Diamond$$

The notion of upward-compatibility corresponds to Milner's weak simulation relation (see for example [Mil04, p. 53]). In a weak simulation, larger states can imitate the behaviour of smaller ones via (multiple) internal actions and one visible action. Here, we consider unlabelled transition systems and identify our $\rightarrow$ with Milner's $\xrightarrow{\tau}$. This imitation of behaviour allows for transitive closure of the transition relation as stated in the following lemma.

[3] The fact that the WQO has to be decidable closes the doors for systems such as Turing machines.

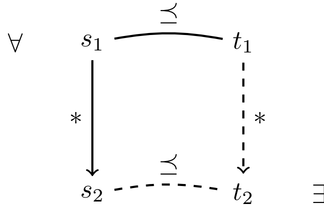


Figure 2.4.: Upward-compatibility of $\rightarrow$ for WSTSs.

Lemma 2.4 (Transitivity of Upward-Compatibility). Given a well-structured transition system $(S, \rightarrow, \preceq)$, the upward-compatibility holds for the reflexive transitive closure of the transition relation, i.e.

$$\forall s_1, s_2, t_1 \in S : (s_1 \preceq t_1 \wedge s_1 \rightarrow^* s_2 \Rightarrow \exists t_2 \in S : s_2 \preceq t_2 \wedge t_1 \rightarrow^* t_2) \,.$$

Proof. Consider WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$ and $t_1, s_1, s_2, \ldots, s_n \in S$ with $s_1 \preceq t_1$ and $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n$, so $s_1 \rightarrow^* s_n$. By Def. 2.6 (WSTS) for every $i > 0$ there is a t_{i+1} s.t. $t_i \rightarrow^* t_{i+1}$ and $s_{i+1} \preceq t_{i+1}$. □

Upward-compatibility is presented diagrammatically in Fig. 2.4 where we quantify existentially over dashed lines and universally over solid lines [FS01]. Intuitively, the WQO property together with upward-compatibility forces every infinite sequence of states into a useful form of repetition (where states may grow).

2.4. The Coverability Problem

We are interested in the coverability problem for WSTSs. Coverability is a variant of the reachability problem that comes with two advantages. It is often sufficient for verification in practice and it remains decidable for many models where reachability becomes undecidable.
For the definition of coverability, we need upward-closed sets of states.

Definition 2.7 (Upward Closed Set). Given a QO $\preceq$ over S, an *upward-closed set* (UCS) is a set $X \subseteq S$ s.t. $\forall x \in X\ \forall s \in S : (x \preceq s \Rightarrow s \in X)$. To any $Y \subseteq S$ we denote its upward-closure $\{\, s \in S \mid \exists y \in Y : y \preceq s \,\}$ with $\uparrow Y$. A *basis* of a UCS X is a set basis(X) s.t. $X = \bigcup_{x \in \text{basis}(X)} \uparrow \{\, x \,\}$.

The closure adds all states that dominate elements in X according to the QO. A set $X \subseteq S$ is *upward-closed* if it satisfies $X = \uparrow X$.
If the set under the upward-closure operator is a singleton, we may omit the curly braces: $\uparrow x = \uparrow \{ x \}$. ◊

In Lemma 2.6, we will learn that any UCS has a finite basis and therefore use only finite bases in this work.

Lemma 2.5 (Properties of Upward Closure). Given a QO $\preceq$ over some set S and sets $X, Y \subseteq S$, the following properties hold.

1. $\uparrow X = \uparrow\uparrow X$ (Idempotence)
2. $\uparrow(X \cup Y) = \uparrow X \cup \uparrow Y$ (Distributivity over $\cup$)
3. $X \subseteq Y \Rightarrow \uparrow X \subseteq \uparrow Y$ (Monotonicity)

Proof. Let $\preceq$ a QO over some set S and sets $X, Y \subseteq S$.

1. $\uparrow\uparrow X = \{ s \in S \mid \exists x \in \uparrow X : x \preceq s \}$
   $= \{ s \in S \mid \exists x' \in \{ s' \in S \mid \exists x \in X : x \preceq s' \} : x \preceq s \}$
   $= \{ s \in S \mid \exists x \in X \; \exists x' \in S : x \preceq x' \preceq s \}$
   $= \{ s \in S \mid \exists x \in X : x \preceq s \}$
   $= \uparrow X$
2. $\uparrow(X \cup Y) = \{ s \in S \mid \exists x \in (X \cup Y) : x \preceq s \}$
   $= \{ s \in S \mid (\exists x \in X : x \preceq s) \vee (\exists y \in Y : y \preceq s) \}$
   $= \{ s \in S \mid \exists x \in X : x \preceq s \} \cup \{ s \in S \mid \exists y \in Y : y \preceq s \}$
   $= \uparrow X \cup \uparrow Y$
3. Let $X \subseteq Y$. We decompose Y and use distributivity of $\uparrow$ over $\cup$.
   $\uparrow Y = \uparrow(X \cup (Y \setminus X)) = \uparrow X \cup \uparrow Y \setminus X$
   Hence, $\uparrow X$ is a subset of $\uparrow Y$. □

Example 2.3. Turning to our running example of PNs and vectors of natural numbers, Dickson's Lemma [Dic13] can be formulated equivalently to express that for any subset of $\mathbb{N}^P$ there are finitely many minimal elements w.r.t. $\leq$. We can identify the minimal vectors of the infinite set

$$U = \uparrow \left\{ (1,0,0,0,0,0,n)^\mathsf{T}, (0,1,2,0,0,3+n,2n)^\mathsf{T} \mid n \in \mathbb{N} \right\}$$

quite easily: We choose $n = 0$ and ensure that there is no pair of vectors comparable w.r.t. the well-quasi ordering. The set of minimal vectors is

$$B = \left\{ (1,0,0,0,0,0,0)^\mathsf{T}, (0,1,2,0,0,3,0)^\mathsf{T} \right\} .$$

It is a (finite and minimal) basis of the former set, i.e. the upward-closure of the basis B coincides with the UCS U:

$$\begin{aligned} &\uparrow \left\{ (1,0,0,0,0,0,0)^\mathsf{T}, (0,1,2,0,0,3,0)^\mathsf{T} \right\} \\ =&\uparrow \left\{ (1,0,0,0,0,0,n)^\mathsf{T}, (0,1,2,0,0,3+n,2n)^\mathsf{T} \mid n \in \mathbb{N} \right\} . \qquad \Diamond \end{aligned}$$

Upward-closed sets—in particular a representation by *finite* bases—are the foundation for the underlying data structures for the implementation (cf. Ch. 6 on p. 160). The following lemma shows that any UCS can be described finitely.

Lemma 2.6 (Finite Basis). If $\preceq$ is a WQO, then any UCS has a finite basis. [Hig52, Theorem 2.1 (i)]

Proof. Consider WQO $\preceq$ over X, some UCS $I \subseteq X$ and some set of minimal elements of I, $I' = \{ x \in I \mid \forall y \in I : (y \preceq x \Rightarrow x = y) \}$. Set I' is a basis of I as $\preceq$ is well-founded. Furthermore, I' *cannot be infinite*, otherwise it would form an infinite sequence of incomparable elements contradicting the WQO property. $\square$

There can be several different sets of minimal elements of a UCS as the WQO is not antisymmetric by definition. There may exist elements s.t. $x \leq y \leq x$ in spite of $x \neq y$.

As mentioned before, we restrict ourselves to the use of finite bases. A finite basis of set X is a subset $\text{basis}(X)$ of minimal elements that are pairwise incomparable (w.r.t. the WQO). By minimality of the elements in the basis, we have $\uparrow \text{basis}(X) = \uparrow X$.[4]

To define the coverability problem, we also need the notion of covering transition sequences.

[4]Note that $\uparrow Y = \uparrow X$ holds for any set Y with $\text{basis}(X) \subseteq Y \subseteq \uparrow X$.

Definition 2.8 (Coverability Relation and Covering Predecessors). Given a WSTS $(S, \to, \preceq)$ and sets of states $X, Y \subseteq S$, by $X \hookrightarrow Y$ we denote the fact that there is a (possibly empty) transition sequence from state X to a state in $\uparrow Y$. Formally,

$$X \hookrightarrow Y :\Leftrightarrow \exists x \in X, y \in \uparrow Y : x \to^* y\,.$$

We say that Y is *coverable from* X and that X contains *covering predecessors* of Y. ♢

If one of the sets is a singleton, we may omit the curly braces. For example, we would write $x \hookrightarrow Y$ instead of $\{\, x \,\} \hookrightarrow Y$.

Example 2.4. In Fig. 2.2b, final marking $m_\Omega = (0,1,0,0,0,0,2)^\mathsf{T}$ for N_{ex} is shown. Marking m_Ω is coverable from initial marking $m_\alpha = (1,1,0,0,0,0,0)^\mathsf{T}$ which is depicted in Fig. 2.2a. A transition sequence which certifies the coverability of m_Ω from m_α is $(t_3\, t_1\, t_5)^2$ is depicted in Fig. 2.5 together with the reached states. ♢

$$\begin{array}{ccccc}
(1,1,0,0,0,0,0)^\mathsf{T} & \xrightarrow{t_3} & (1,0,1,0,1,0,0)^\mathsf{T} & \xrightarrow{t_1} & (1,1,0,0,1,0,0)^\mathsf{T} \\
= m_\alpha & & & & \downarrow t_5 \\
(1,1,0,0,1,0,1)^\mathsf{T} & \xleftarrow[t_1]{} & (1,0,1,0,1,0,1)^\mathsf{T} & \xleftarrow[t_3]{} & (1,1,0,0,0,0,1)^\mathsf{T} \\
\downarrow t_5 & & & & \\
(1,1,0,0,0,0,2)^\mathsf{T} & & & & \\
\geq m_\Omega & & & &
\end{array}$$

Figure 2.5.: Witness trace for $m_\alpha \hookrightarrow m_\Omega$ in N_{ex}.

The coverability problem takes as input a WSTS $\mathcal{S}$, a finite set of initial states I, and a finite set of final states F (also called the *target states*). The problem is to decide whether some initial state from I can reach $\uparrow F$. Intuitively, this means a state is reachable that covers some state in F. Later in this chapter, in Sect. 2.6, we will refine the coverability problem to go beyond a yes or no question.

Definition 2.9 (Coverability Problem Cov).

Given: WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$ and $I, F \subseteq S$ finite.

Problem: Does $I \hookrightarrow F$ hold?

Equivalently, the coverability problem **Cov** can be formulated as the question

$$\text{“Does } I \cap pre^*(\uparrow F) \neq \emptyset \text{ hold?”}, \tag{2.1}$$

where we use the following definition of the predecessor function.

Definition 2.10 (Predecessors). Given a WSTS $(S, \rightarrow, \preceq)$ and set $X \subseteq S$, the predecessors are those from which X is reachable in one step, i.e,

$$pre(X) := \{\, y \in S \mid \exists x \in X : y \rightarrow x \,\} \ .$$

The reflexive transitive closure of *pre* is pre^* and the transitive closure is pre^+, formally, $pre^*(X) := \bigcup_{i \in \mathbb{N}} pre^i(X)$ and $pre^+(X) := \bigcup_{i \in \mathbb{N}} pre^{i+1}(X)$. As usual, the identity is used to define the base case, s.t. $pre^0(X) = X$. If the predecessor function is applied to an upward-closed set, we speak of *covering predecessors.* $\Diamond$

When convenient, we write the (reflexive) transitive closure of *pre* by directly using the transition relation $\rightarrow$ as

$$pre^*(X) = \{\, y \in S \mid \exists x \in X : y \rightarrow^* x \,\} \text{ and}$$
$$pre^+(X) = \left\{\, y \in S \mid \exists x \in X : y \rightarrow^+ x \,\right\} ,$$

respectively. The following lemma gives some properties of the predecessor function which we will use for our correctness proofs.

Lemma 2.7 (Properties of Predecessors). For any WSTS $(S, \rightarrow, \preceq)$ and any sets $X, Y \subseteq S$, following equalities hold:

1. $pre(X \cup Y) = pre(X) \cup pre(Y)$ (Distributivity over $\cup$)
   $pre^*(X \cup Y) = pre^*(X) \cup pre^*(Y)$
   $pre^+(X \cup Y) = pre^+(X) \cup pre^+(Y)$

2. $X \subseteq Y \Rightarrow pre(X) \subseteq pre(Y)$ (Monotonicity)
   $X \subseteq Y \Rightarrow pre^*(X) \subseteq pre^*(Y)$
   $X \subseteq Y \Rightarrow pre^+(X) \subseteq pre^+(Y)$

3. $pre^*(X) = pre^*(pre^*(X))$ (Idempotence of pre^*)
   $pre^*(pre^+(X)) = pre^+(pre^*(X)) = pre^+(X)$

4. $pre^*(X) = X \cup pre^*(pre(X))$ (Expansion of pre^*)

5. $pre^+(X) = pre(X) \cup pre^+(pre(X))$ (Expansion of pre^+)

6. $pre^*(X) = X \cup pre^+(X)$ (Relationship of pre^+ and pre^*)
   $pre^+(X) = pre^*(pre(X)) = pre(pre^*(X))$

Proof. Let $(S, \rightarrow, \preceq)$ a WSTS and sets $X, Y \subseteq S$. We closely follow the definitions of pre, pre^* and pre^+.

1. $pre(X \cup Y) = \{ z \in S \mid \exists x \in (X \cup Y) : z \rightarrow x \}$
   $= \{ z \in S \mid (\exists x \in X : z \rightarrow x) \vee \exists y \in Y : z \rightarrow y) \}$
   $= \{ z \in S \mid \exists x \in X : z \rightarrow x \} \cup \{ z \in S \mid \exists y \in Y : z \rightarrow y \}$
   $= pre(X) \cup pre(Y)$
   Proofs for pre^* and pre^+ are analogous.

2. Let $X \subseteq Y$. We decompose Y and use distributivity of pre over $\cup$.
   $pre(Y) = pre(X \cup (Y \setminus X)) = pre(X) \cup pre(Y \setminus X)$
   Hence, $pre(X)$ is a subset of $pre(Y)$. Proofs for pre^* and pre^+ are analogous.

3. $pre^*(pre^*(X)) = \bigcup_{i \in \mathbb{N}} pre^i(pre^*(X))$
   $= \bigcup_{i \in \mathbb{N}} pre^i(\bigcup_{j \in \mathbb{N}} pre^j(X))$
   $= \bigcup_{i,j \in \mathbb{N}} pre^i(pre^j(X))$
   $= \bigcup_{i,j \in \mathbb{N}} pre^{i+j}(X)$
   $= \bigcup_{k \in \mathbb{N}} pre^k(X)$
   $= pre^*(X)$
   Proof for $pre^*(pre^+(X)) = pre^+(pre^*(X)) = pre^+(X)$ is analogous.

4. $pre^*(X) = \bigcup_{i \in \mathbb{N}} pre^i(X)$
   $= pre^0(X) \cup \bigcup_{i \in \mathbb{N}} pre^{i+1}(X)$
   $= X \cup \bigcup_{i \in \mathbb{N}} pre^i(pre(X))$
   $= X \cup pre^*(pre(X))$

5. $pre^+(X) = \bigcup_{i \in \mathbb{N}} pre^{i+1}(X)$
   $= pre^1(X) \cup \bigcup_{i \in \mathbb{N}} pre^{i+2}(X)$
   $= pre(X) \cup \bigcup_{i \in \mathbb{N}} pre^{i+1}(pre(X))$
   $= pre(X) \cup pre^+(pre(X))$

6. $pre^+(X) = \bigcup_{i\in\mathbb{N}} pre^{i+1}(X) = \bigcup_{i\in\mathbb{N}} pre^i(pre(X))$
   $= \bigcup_{i\in\mathbb{N}} pre(pre^i(X))$ and
   $pre^*(X) = \bigcup_{i\in\mathbb{N}} pre^i(X) = pre^0(X) \cup \bigcup_{i\in\mathbb{N}} pre^{i+1}(X)$
   $= X \cup pre^+(X)$ □

The following lemma uses the upward-compatibility of WSTSs to provide some insight in properties of the predecessor function.

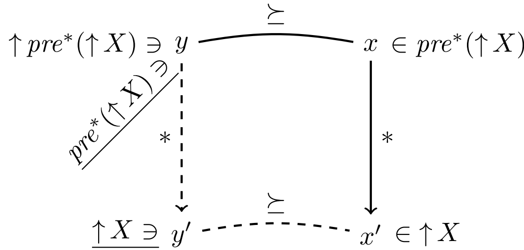


Figure 2.6.: Upward-compatibility in the proof of Lemma 2.8.

Lemma 2.8 (Predecessors of Upward Closed Sets). Given a WSTS $(S, \rightarrow, \preceq)$ and set $X \subseteq S$, the reflexive transitive closure of predecessors of an upward-closed set is upward-closed:

$$pre^*(\uparrow X) = \uparrow pre^*(\uparrow X)$$

Proof. Let $(S, \rightarrow, \preceq)$ a WSTS and $X \subseteq S$ a set of states.

1. Show $pre^*(\uparrow X) \subseteq \uparrow pre^*(\uparrow X)$: This holds by definition of the upward-closure (cf. Def. 2.7).

2. Show $\uparrow pre^*(\uparrow X) \subseteq pre^*(\uparrow X)$: Let $y \in \uparrow pre^*(\uparrow X)$. From the definition of pre^* and $\uparrow$, it follows that there exist $x \in pre^*(\uparrow X), x' \in \uparrow X$, s.t.
   $$y \succeq x \rightarrow^* x' \in \uparrow X \,.$$
   By upward-compatibility of WSTSs (cf. Def. 2.6), we know that state y' exists with $y \rightarrow^* y' \succeq x'$. As y' covers x', y' too is in the upward-closure of X. Therefore, y must belong to $pre^*(\uparrow X)$. A graphical representation is given in Fig. 2.6. Dashed lines represent

guarantees by the upward-compatibility of the WQO with $\rightarrow$ and underlined sets indicate deductions via upward-compatibility.

We conclude that $pre^*(\uparrow X) = \uparrow pre^*(\uparrow X)$ holds. $\square$

The following lemma captures a central property of upward-closed sets that is used throughout our work.

Lemma 2.9 (Stabilization). For a WQO $\preceq$ over some set X, any infinite monotone sequence $I_0 \subseteq I_1 \subseteq \cdots$ of UCSs eventually stabilizes, i.e. $\exists k \in \mathbb{N}\, \forall i \in \mathbb{N} : I_k = I_{k+i}$. [Hig52, Theorem 2.1 (ii)]

Proof. Consider a counter-example, i.e. an infinite monotone sequence with $\forall k \in \mathbb{N}\ \exists i \in \mathbb{N} : I_k \neq I_{k+i}$. We extract an infinite subsequence with strict inclusion $I_{n_0} \subset I_{n_1} \subset \cdots$. For any $i > 0$ we select some $x_i \in I_{n_i} \setminus I_{n_{i-1}}$ and conclude by the WQO property that the infinite sequence $x_1, x_2, \ldots$ contains indices $j < k$ s.t. $x_j \preceq x_k$. Since $x_j \in I_{n_j}$ and I_{n_j} is a UCS, $x_k \in I_{n_j}$, contradicting $x_k \notin I_{n_{k-1}}$. Thus, the assumed counter-example does not exist. $\square$

As a first application of the stabilization lemma, we show that pre^* stabilizes, i.e. a finite union over pre^i suffices when the set of states under consideration is upward-closed.

Lemma 2.10 (Stabilization of UCS-Predecessors). Given a WSTS $(S, \rightarrow, \preceq)$ and set $X \subseteq S$, a basis of the (upward-closed) reflexive transitive closure of predecessors of an upward-closed set is reached after a finite number of steps:

$$\exists k \in \mathbb{N}\, \forall j \in \mathbb{N} : \uparrow \bigcup_{i=0}^{k+j} pre^i(\uparrow X) = pre^*(\uparrow X)$$

Proof. Let $(S, \rightarrow, \preceq)$ a WSTS and $X \subseteq S$ a set of states. Observe, that

$$\uparrow \bigcup_{i=0}^{0} pre^i(\uparrow X) \subseteq \uparrow \bigcup_{i=0}^{1} pre^i(\uparrow X) \subseteq \uparrow \bigcup_{i=0}^{2} pre^i(\uparrow X) \subseteq \cdots$$

is an infinite monotone sequence of upward-closed sets, as inclusion

$$\bigcup_{i=0}^{n} pre^i(\uparrow X) \subseteq \bigcup_{i=0}^{n+1} pre^i(\uparrow X)$$

holds and the upward-closure operator is monotone (Lemma 2.5-3).
By the stabilization lemma (Lemma 2.9), there is a natural k, s.t. for any $j \in \mathbb{N}$:

$$\uparrow \bigcup_{i=0}^{k} pre^i(\uparrow X) = \uparrow \bigcup_{i=0}^{k+j} pre^i(\uparrow X)\,.$$

Henceforth, we deduce that for any $j \in \mathbb{N}$

$$\uparrow \bigcup_{i=0}^{k+j} pre^i(\uparrow X) = \uparrow \bigcup_{i=0}^{\infty} pre^i(\uparrow X)$$

holds. From Lemma 2.8 we know that the set of predecessors of a UCS is a UCS itself, i.e. $pre^*(\uparrow X) = \uparrow pre^*(\uparrow X)$. By definition of pre^*, this leads to the equality of $pre^*(\uparrow X)$ and $\uparrow \bigcup_{i=0}^{\infty} pre^i(\uparrow X)$ which concludes the proof. □

Example 2.5. In N_{ex}, the predecessors of set $\uparrow m_\Omega$ are

$$pre(\uparrow m_\Omega) = \uparrow \left\{ (0,0,1,0,0,0,2)^\mathsf{T}, (1,2,0,0,0,0,2)^\mathsf{T}, (0,1,0,0,1,0,1)^\mathsf{T} \right\}.$$

The three elements of the minimal basis are covering predecessors w.r.t. the transitions t_1, t_2 and t_5, meaning that their successors w.r.t. the transitions cover marking m_Ω: For any $m \in \mathbb{N}^P$ we have

$$\begin{aligned}
(0,0,1,0,0,0,2)^\mathsf{T} + m &\xrightarrow{t_1} (0,1,0,0,0,0,2)^\mathsf{T} + m \geq m_\Omega \\
(1,2,0,0,0,0,2)^\mathsf{T} + m &\xrightarrow{t_2} (0,1,0,1,0,0,2)^\mathsf{T} + m \geq m_\Omega \\
(0,1,0,0,1,0,1)^\mathsf{T} + m &\xrightarrow{t_5} (0,1,0,0,0,0,2)^\mathsf{T} + m \geq m_\Omega\,.
\end{aligned}$$

Therefore, markings $(0,0,1,0,0,0,2)^\mathsf{T} + m$, $(1,2,0,0,0,0,2)^\mathsf{T} + m$, and $(0,1,0,0,1,0,1)^\mathsf{T} + m$ are in $pre(m_\Omega)$. One-step predecessors w.r.t. the

other transitions are

$$(0,1,1,0,0,0,2)^{\mathsf{T}} + m \xrightarrow{t_3} (0,2,0,0,0,0,2)^{\mathsf{T}} + m \geq m_\Omega$$
$$(0,1,0,1,1,0,2)^{\mathsf{T}} + m \xrightarrow{t_4} (0,1,0,1,0,1,2)^{\mathsf{T}} + m \geq m_\Omega\,.$$

These predecessors are already contained in the upward-closure of the predecessors w.r.t. t_1, t_2 and t_5. Hence, they do not show up in the minimal basis.

Note that the predecessors w.r.t. t_2, t_3 and t_4 each cover m_Ω. In practice, such predecessors are superfluous and can be safely ignored. We will discuss this property in Sect. 2.5 and comment again in the context of building our framework in Sect. 3.2 on p. 69, more precisely, we discuss it together with Alg. 3.2 on p. 70. $\Diamond$

As we will learn, the coverability problem is decidable if the WSTS under consideration possesses an effectively computable *pred-basis*.

Definition 2.11 (Pred-Basis). Let $(S, \rightarrow, \preceq)$ be a WSTS and X a subset of S. By $pb(X) := \text{basis}(pre(\uparrow X))$ we denote a basis for the set of one-step predecessors (*pred-basis*) of the upward-closure of X. $\Diamond$

As a consequence, we only consider WSTSs for which pb is effectively computable.

In Example 2.5 we have already given an example for the pred-basis $pre(\uparrow m_\Omega)$ of $\{ m_\Omega \}$.

In contrast to upward-closed sets, we define downward-closed sets by finite bases. While downward-closed sets have their place in the procedures to analyse well-structured transition systems, they are not used as prominently as upward-closed sets.

Definition 2.12 (Downward-Closed Set). Given a QO $\preceq$ over X and some finite set J', an *downward-closed set* (DCS) is a set $J \subseteq X$ which contains all elements that are less than or equal to an element in J' w.r.t. QO, i.e. $J = \downarrow J' = \{ x \in X \mid \exists y \in J' : x \preceq y \}$. Set J' is the *finite basis* of J. $\Diamond$

2.5. Decidability Results

A main result in the theory of WSTSs is the decidability of the coverability problem **Cov** via a *backward reachability analysis* (BR)[5]. The corresponding and by now standard *backward algorithm* stems from [AČJT96]. It performs a fixed-point iteration to compute the transitive closure of the predecessor relation when starting at the final states. Recently, it has been shown that this approach is optimal for Petri nets w.r.t. the complexity of the coverability problem [BG11].

Input : WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$, finite set of states $F \subseteq S$
Output : A set $V \subseteq S$, s.t. $\uparrow V = pre^*(\uparrow F)$
Comment: Given finite base F, the algorithm computes a finite base V of backward reachable states.

```
1  W := F;
2  V := ∅;
3  while W ≠ ∅ do
                                      /* States to process in W. */
4      x := select(W) ;           /* Select some x from basis W. */
5      W := W \ { x } ;             /* Remove x from basis W. */
6      if x ∉ ↑V then
                                         /* x not yet processed. */
7          V := V ∪ { x } ;                 /* Add x to basis V. */
8          W := W ∪ pb(x)    /* Add x's predecessors to W. */
9      fi
10 od
                                        /* All states processed. */
```

Algorithm 2.1: Basic backward reachability analysis Basic BR.

In Alg. 2.1, we present the basic algorithm to discuss the core mechanisms of BR. The selection function used in this algorithm has to satisfy a single property.

[5] While the analysis decides coverability problems, it is called backward *reachability* analysis for historic reasons. We use both terms interchangeably.

Definition 2.13 (Selection Function). Function *select* is admissible if it returns a single element when given a non-empty set. Formally, for any set W,

$$W \neq \emptyset \Rightarrow \mathit{select}(W) \in W\,. \qquad \lozenge$$

As input, the algorithm takes a WSTS and a finite set of final states F. It computes a finite base V of all states that are backward reachable from the upward-closure of F, i.e. $\mathit{pre}^*(\uparrow F)$. The result V can thus be used to answer the coverability problem **Cov** via the formulation of Eq. 2.1: Let I be a finite set of initial states and F the finite basis of final states.

"The answer to coverability problem $I \cap \mathit{pre}^*(\uparrow F) \neq \emptyset$ is $I \cap \uparrow V \neq \emptyset$."

As both I and V are finite, we may formulate it operationally:

"The answer to coverability problem $I \cap \mathit{pre}^*(\uparrow F) \neq \emptyset$ is $\exists s \in I, t \in V : s \succeq t$."

The algorithm works as follows. Final states F are interpreted as a finite basis W of a UCS which contains all states that are backward reachable from $\uparrow F$ and have to be explored further. A second UCS of states backward reachable from $\uparrow F$ which are already fully explored is represented by the finite basis V.
In each iteration a single state x—standing for a whole UCS $\uparrow x$—is removed from W. If all the states in $\uparrow x$ are fully explored, that is $\uparrow x \subseteq \uparrow V$ or simply $x \in \uparrow V$, the next iteration is commenced. If $\uparrow x$ is not fully explored yet, i.e. $x \notin \uparrow V$, it is added to V and a finite basis $\mathit{pb}(x)$, describing the one-step predecessors of $\uparrow x$, is added to W. When there are no more states to explore, the loop condition $W \neq \emptyset$ becomes false and $\uparrow V$ is the set of states backward reachable from $\uparrow F$, $\mathit{pre}^*(\uparrow F)$.
Note that when predecessors of x are added to W, state x is guaranteed to be in $\uparrow V$ as it has just been added to V. As hinted in Example 2.5, predecessors of x which cover x can be safely ignored: since $x \in \uparrow V$ each such predecessor of x will be removed from W without further action.

Example 2.6 (Coverable Final State). In Fig. 2.7, we have drawn the search space of N_{ex} w.r.t. m_Ω in a condensed way. Markings are written in multiset notation and predecessors of states that immediately cover that respective state are omitted. Each path from a marking that

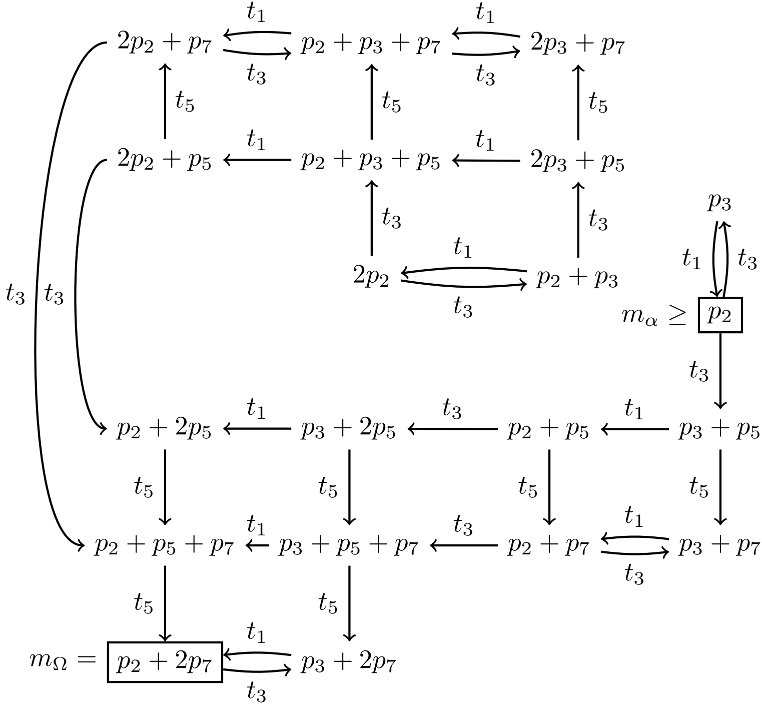


Figure 2.7.: Backward exploration of N_{ex}, starting in m_Ω.

is covered by initial marking m_α to marking m_Ω represents a witness transition sequence. For example, in the graph, the path from p_2, the only marking covered by $m_\alpha = p_1 + p_2$, via transitions $(t_3\, t_1)^3\, (t_3\, t_1\, t_5)^2\, (t_3\, t_1)^4$ leads to some marking which covers m_Ω. Indeed, the marking reached by firing this transition sequence from m_α is $p_1 + p_2 + 7p_5 + 2p_7$.

The subgraph at the top does not contain a state covered by the initial marking. None of the markings of that subgraph can be covered from m_α as no transition sequence leads into that part of the graph. The backward analysis is obviously able to explore states that are not part of the system's state space. We will discuss pruning techniques to alleviate this effect in Sect. 5.2 on p. 143.

Here, arrows do not represent the transition relation. They are explored backwards and lead to the minimal basis of covering predecessors w.r.t. a single transition. ♢

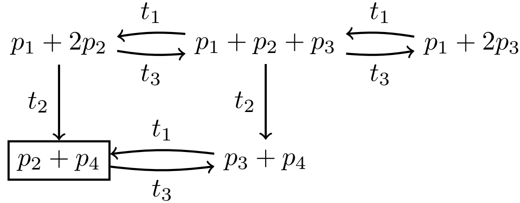


Figure 2.8.: Backward exploration of N_{ex}, starting in $p_2 + p_4$.

Example 2.7 (Uncoverable Final State). In Fig. 2.8, we have drawn the search space of N_{ex} w.r.t. marking $p_2 + p_4$ in a condensed way. This marking is not coverable from m_α. Again, markings are written in multiset notation and predecessors of states that immediately cover that respective state are omitted and arrows do not represent the transition relation. They are explored backwards and lead to the minimal basis of covering predecessors w.r.t. a single transition.
As no state covered by the initial marking is contained in the graph, it poses as a witness for the uncoverability of $p_2 + p_4$. ♢

Observe that no states are ever removed from V. Therefore, the number of loop iterations is either finite or the sequence $\uparrow V_0 \subseteq \uparrow V_1, \uparrow V_2 \subseteq \ldots$ of UCSs represented by the content of V in each loop iteration forms an infinite monotone sequence. The stabilization lemma (Lemma 2.9) provides insight in why the algorithm is bound to terminate: The infinite monotone sequence of upward-closed sets $\uparrow V_0 \subseteq \uparrow V_1, \uparrow V_2 \subseteq \ldots$ stabilizes in a finite number of steps.
As an informal argument for correctness, consider equality

$$\uparrow V \cup pre^*(\uparrow W) = pre^*(\uparrow F)$$

which is an invariant of the algorithm's loop. It is fulfilled at loop entry with $W = F$ and $V = \emptyset$. At the end of the loop's body it is satisfied because every state that is removed from W is ensured to be contained in V. The only way for a state to enter the algorithm is by pb. After

termination of the loop, W is empty and the invariant collapses to $\uparrow V = pre^*(\uparrow F)$.
We prove this formally in Sect. 3.1 on p. 52.
It is Alg. 2.1 that we generalize and optimize in the following chapters of this work. Our goal is to answer quickly to common and positive instances of the coverability problem.

2.6. Well-Structured Labelled Transition Systems

As stated in Sect. 2.4, we want to go further than the question whether a set of states is coverable. We want to know *how* it is coverable.
To be able to identify transition sequences, we turn to the definition of labelled transition systems and extend WSTSs correspondingly. Upward-compatibility in the context of well-structured labelled transition systems is depicted in Fig. 2.9.

Definition 2.14 (Well-Structured Labelled Transition System).

A structure $\mathcal{S} = (S, L, \rightarrow, \preceq)$ is a *well-structured labelled transition system* (WSLTS) if S is a set of states, $\rightarrow : S \times L \times S$ is a transition relation, $\preceq$ is a decidable WQO, such that the transition relation is *consistent* and upward-compatible with $\preceq$, i.e.

$$\forall s_1, s_2, t_1 \in S\ \forall a \in L : (s_1 \preceq t_1 \wedge s_1 \xrightarrow{a} s_2 \Rightarrow \exists t_2 \in S : s_2 \preceq t_2 \wedge t_1 \xrightarrow{a} t_2)\,.$$

By $s_0 \xrightarrow{a_1 a_2 \cdots a_k} s_k$, we denote the existence of a transition sequence from s_0 to s_k labelled with $a_1 a_2 \cdots a_k \in L^*$, i.e. $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{k-1}} s_{k-1} \xrightarrow{a_k} s_k$. $\Diamond$

The predecessor function (cf. Def. 2.10) can remain unchanged as it disregards labels.
We will use the two definitions of WSTSs and WSLTSs synonymously and distinguish only when it is not clear from context.

Definition 2.15 (Labelled Coverability Relation). Consider well-structured labelled transition system $(S, L, \rightarrow, \preceq)$, sets of states $X, Y \subseteq S$, and a finite sequence $\sigma \in L^*$ of transition labels. By $X \overset{\sigma}{\hookrightarrow} Y$ we

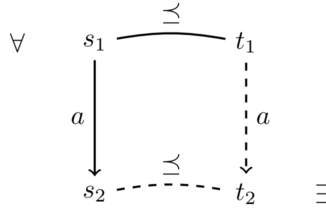


Figure 2.9.: Upward-compatibility of $\rightarrow$ for WSLTSs.

denote the fact that there is a *transition sequence* from a state in X to a state in $\uparrow Y$ that is labelled by σ. Formally,

$$X \overset{\sigma}{\hookrightarrow} Y :\Leftrightarrow \exists x \in X, y \in \uparrow Y : x \xrightarrow{\sigma} y\,.$$

We write $X \hookrightarrow Y$ if there is $\sigma \in L^*$ for which $X \overset{\sigma}{\hookrightarrow} Y$ holds. ◊

If one of the sets is a singleton, we may omit the curly braces. For example, we would write $x \overset{\sigma}{\hookrightarrow} Y$ instead of $\{x\} \overset{\sigma}{\hookrightarrow} Y$.

The labelled coverability problem we are concerned with takes as input a WSLTS $\mathcal{S}$, a finite set of initial states I, and a finite set of final states F. The problem is to decide whether some state in $\uparrow F$ can be reached from a state in I. Intuitively, this means that a state is reachable which covers some state in F. To be useful for verification, the coverability analysis should also come up with a transition sequence that leads to this covering state.

The running Petri net example and both the example search spaces Example 2.6 and Example 2.7 of Alg. 2.1 already make use of transition labels.

Definition 2.16 (Labelled Coverability Problem LCov).

Given: WSLTS $\mathcal{S} = (S, L, \rightarrow, \preceq)$ and $I, F \subseteq S$ finite.

Problem: Does $I \hookrightarrow F$ hold? In that case, determine (x, σ) with $x \in I$ and $\sigma \in L^*$ s.t. $x \overset{\sigma}{\hookrightarrow} F$.

Clearly, a solution to **Cov** can be generated from a solution to **LCov** as it contains *more* information in case $I \hookrightarrow F$ holds.

While this problem can be solved by a variation of the basic backward reachability analysis (Alg. 2.1), we strive to generalize the algorithm to a framework which can be instantiated easily and provides optimizations independent of the concrete system class, as well as efficient data structures out of the box.

2.7. Instances of WSLTSs

The importance of WSLTSs stems from the fact that they capture in a uniform way Petri nets, lossy channel systems, and other seemingly different formalisms. Since some of our optimizations are specific to certain classes of WSLTSs, we introduce them in more detail.

2.7.1. Petri Nets

The class of P/T Petri nets we introduced in Sect. 2.1 is often used to model concurrent processes such as business processes and parallel programs where mutual exclusion is of interest. Note that mutual exclusion fails if *at least two programs* enter some critical section. Hence, in the corresponding correctness proof one ensures that no marking m' is reachable that dominates marking m where two programs are in the critical section.

For PNs, a coverability problem asks for a transition sequence from some initial marking m_0 to a marking which puts at least as many tokens on places as some final marking does. In this setting, the problem is known to be EXPSPACE-complete [Rac78, Lip76].

There are many extensions of Petri nets available, one of which we will consider in Sect. 2.7.2. Other extensions, such as inhibitor arcs or timed Petri nets, often introduce Turing-completeness or make use of true concurrency semantics. We are not concerned with those extensions.

Example 2.8 (PN Example: Mutual Exclusion). In the Petri net of Fig. 2.10, two processes that synchronize on a shared resource, the token of place p_5, are modelled where places p_2 and p_4 represent activities that cannot not occur at the same time. Mutual exclusion of these activities is forced by the single token on place p_5. When a process fires a transition (t_1 or t_3 respectively) to enter its critical activity, it takes the token from p_5 and disables the other process's transition, prohibiting it from entering

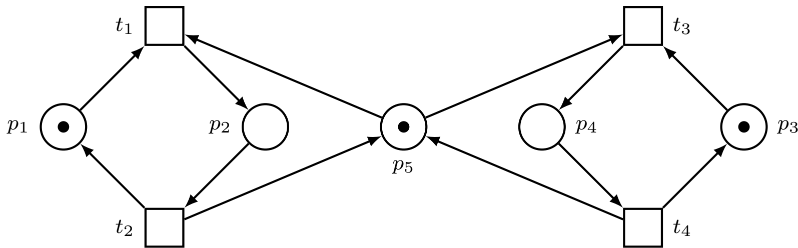


Figure 2.10.: P/T Petri net example: Mutual exclusion of p_2 and p_4.

its critical activity. When the first process leaves its critical activity by firing the only enabled transition of the net, a token is created on p_5 and the whole process starts anew. $\Diamond$

2.7.2. Petri Nets With Transfer

Petri nets with transfer (PNT) extend Petri nets by arcs whose weight depends on the current marking. This allows for transitions that empty a place in one step and transitions that add up tokens on different places. The model has been proven useful in the analysis of synchronization skeletons for parametrized and synchronized multi-threaded programs [DRV02, KKW10].

Compared to Petri nets, the additional expressiveness of transfer nets results from an extended weight function. Transfer nets not only have natural numbers as weights but linear polynomials over the places. The following definition is taken from [SM12].

Definition 2.17 (Petri Net With Transfer). A transfer net (PNT) is a triple $N = (P, T, W)$ where $P = \{ p_1, \ldots, p_{|P|} \}$ is a finite set of *places*, T a finite set of *transitions*, and $W : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}[P]$ is a *weight function*. Thus, the weights are polynomials over the places with natural-valued coefficients. The weight function satisfies two conditions:

1. The weight of an arc from transition t to place p, $W(t, p)$, has the form $k + \sum_{i=1}^{|P|} \lambda_i p_i$, where the λ_i and k are natural numbers.

2. The weight of an arc from place p to transition t, $W(p, t)$, is either p itself (a *reset arc*) or a natural number (a *classical arc*).

Given a marking m and $(x, y) \in (P \times T) \cup (T \times P)$, we denote by $W(x, y)(m)$ the application of function $W(x, y)$ to m: if $W(x, y)$ is $k + \sum_{i=1}^{|P|} \lambda_i p_i$ then the application to m is $k + \sum_{i=1}^{|P|} \lambda_i m(p_i)$. This includes the special cases of reset and classical arcs.
In this extended class of Petri nets, a transition is enabled in marking m, if $m \geq W(-, t)(m)$. The result of firing an enabled transition t from a marking m_1 to marking m_2 now is defined as $m_2 = m_1 - W(-, t)(m_1) + W(t, -)(m_1)$. ♢

This class of Petri nets matches so-called reset post self-modifying nets [DFS98], a special case of Valk's self-modifying nets introduced in [Val78a, Val78b]. Our definition closely relates to [DFS98][6] and subsumes reset nets, so that the coverability problem is of non-primitive recursive complexity [Sch10]. Among others, this class of extended Petri nets is also investigated in [Cia94].

Example 2.9 (Buffered Consumer / Producer). In Fig. 2.11 a PNT modelling a buffered producer-consumer system is shown. The system consists of a producer process (places and transitions labels with p) and a consumer process (labels with c) that share two resources: a place *chan* representing a channel between the processes and a place *mutex* to enable mutual exclusion for accessing the channel. Both processes contain internal buffers p_{buffer} and c_{buffer} which are used to minimize the interactions on the shared channel.
The system works as follows.
The producer may create arbitrarily many tokens on its internal buffer. When it decides to transfer the tokens from its buffer to the channel, it first acquires the mutex token and then shifts all tokens from its buffer to the channel in one step. This is expressed by transition p_{send} which takes all tokens from p_{buffer} and transfers them to place *chan* as is described by the input arc from p_{buffer} with weight $1 \cdot p_{buffer}$ (remove all tokens) and the output arc to *chan* with weight $1 \cdot p_{buffer}$ (add as many tokens). After the transfer, the producer releases the acquired mutex token and loops.
The consumer works analogously to the producer. It may process and consume as many tokens as it has stored in its internal buffer. When

[6] Dufourd et al. show that coverability is decidable even for reset post G-nets (generalized reset post self-modifying), a Petri net class that allows for not only linear, but arbitrary polynomials over the places on output arcs.

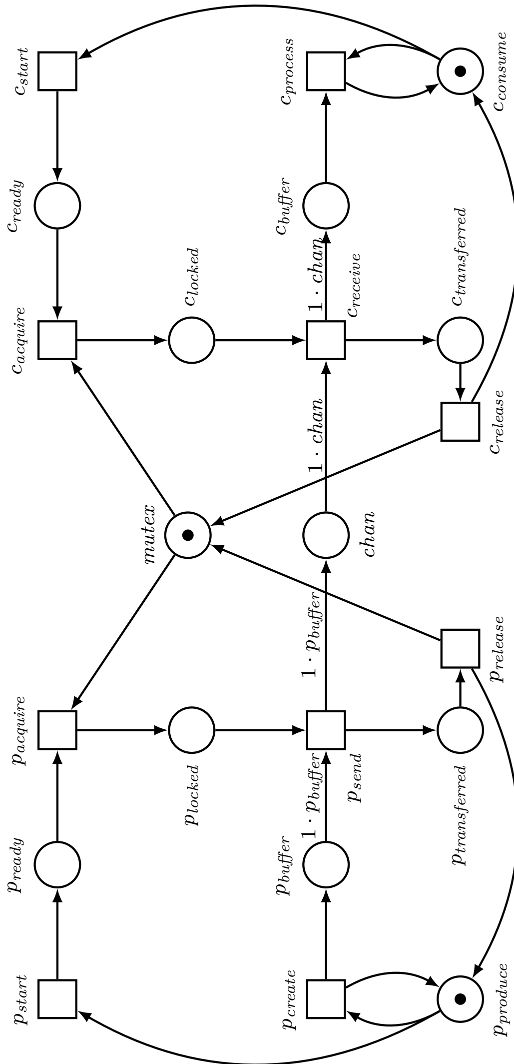


Figure 2.11.: Petri net with transfer example:
Buffered Producer / Consumer.

it decides to transfer the channel's contents to its buffer, it will acquire the mutex token and move all tokens from *chan* to c_{buffer} in one step, followed by release of the mutex token and consummation of the tokens on its buffer. ♢

2.7.3. Lossy Channel Systems

Lossy channel systems formalize network protocols like the alternating bit protocol and more general sliding window protocols. Technically, lossy channel systems are finite state programs that communicate via asynchronous message transfer over unbounded FIFO channels. The restriction that yields decidability is inspired by the following observation about the application domain. Network protocols are designed to operate correctly in the presence of package loss. Therefore, a weaker model with unreliable channels should be sufficient for their verification. Lossy channel systems formalize unreliability by lossiness: channels may drop packages at any moment. They have been studied extensively in the work of Abdulla et al. [AJ93, AJ94, AJ96, AKP97]. We recall the definition of lossy channel systems from [SM12].

Definition 2.18 (Lossy Channel System). A lossy channel system (LCS) is a tuple $L = (Q, C, M, \to)$ where Q is a finite set of *control states* and C is a finite set of *channels* over which we transfer *messages* in the finite set M. *Transitions* in $\to \subseteq Q \times OP \times Q$ perform *operations* in $OP := (C \times \{\,!, ?\,\} \times M) \cup \{\,\tau\,\}$. A transition $(q_1, op, q_2) \in \to$, typically denoted by $q_1 \xrightarrow{op} q_2$, yields a change in the control state from q_1 to q_2 while performing operation op. A *local operation* τ in OP does not change any channel contents. A *send operation* $c!a$ in OP appends message a to the current content of channel c. A *receive operation* $c?a$ in OP removes message a from the head of channel c. Therefore, the last two operations indeed define a FIFO channel.

The semantics of LCSs relies on the notion of configurations. A *configuration of* L is a pair $\gamma = (q, W) \in Q \times {M^*}^C$ consisting of a control state $q \in Q$ and a vector $W \in {M^*}^C$ that assigns to each channel $c \in C$ a finite word $W(c) \in M^*$.

Transitions change the channel content. We capture this by *update operations* $[c := x]$ with $c \in C$, $x \in M^*$ that operate on vectors of words. Applying the update to a channel content $W \in {M^*}^C$ results

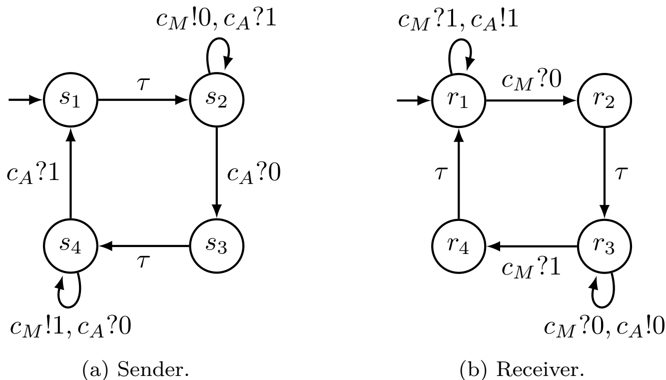
in a new content $W[c := x] \in M^{*C}$ defined by $W[c := x](c) := x$ and $W[c := x](c') := W(c')$ for all $c' \neq c$ with $c' \in C$.
Lossiness is formalized by an ordering on configurations. For the definition, we first compare words by Higman's *subword ordering* [Hig52]. It sets $u \preceq^* v$ if u is a not necessarily contiguous subword of v. More formally, let $u = u_1 \ldots u_m$ and $v = v_1 \ldots v_n$ in M^*. We have $u \preceq^* v$ if there are indices $1 \leq i_1 < \ldots < i_m \leq n$ with $u_j = v_{i_j}$ for all $1 \leq j \leq m$. With a component-wise definition, we lift the ordering to vectors of words, $W_1 \preceq^* W_2$ if $W_1(c) \preceq^* W_2(c)$ for all $c \in C$. For configurations, we pose the additional requirement that the control states coincide. This means, we have $(q_1, W_2) \preceq (q_2, W_2)$ if $q_1 = q_2$ and $W_1 \preceq^* W_2$.
The behaviour of LCSs is defined in terms of transitions between configurations, $\rightarrow \subseteq (Q \times M^{*C}) \times (Q \times M^{*C})$. The transitions are derived with the following rules:

- $(q_1, W) \rightarrow (q_2, W)$ if $q_1 \xrightarrow{\tau} q_2$ (a local operation)
- $(q_1, W) \rightarrow (q_2, W[c := W(c) \cdot m])$ if $q_1 \xrightarrow{c!m} q_2$ (a send operation)
- $(q_1, W[c := m \cdot W(c)]) \rightarrow (q_2, W)$ if $q_1 \xrightarrow{c?m} q_2$ (a receive operation)
- $\gamma_1' \rightarrow \gamma_2'$ if $\gamma_1' \succeq \gamma_1 \rightarrow \gamma_2 \succeq \gamma_2'$ (loss of channel contents)

for some configurations $\gamma_1, \gamma_2 \in Q \times M^{*C}$. $\Diamond$

For LCSs, a coverability problem is the question whether there exists a transition sequence leading from an initial control state q_0 with empty channels to any of the final control states regardless of the channel content, i.e. a configuration is reached with a final control state and a channel content that covers the empty channel content. In this setting, the problem is of non-primitive recursive complexity [Sch02, Sch10].

Example 2.10 (Alternating Bit Protocol [AJ96]). The lossy channel system in Fig. 2.12 models a part of the alternating bit protocol of [BSW69] with the goal that messages are transmitted from the sender to the receiver in the correct order over FIFO channels, although the channels may drop messages non-deterministically. To ensure the messages are delivered in the correct order, sequence numbers are appended and transferred with the messages. In this model, the actual messages



(a) Sender.

(b) Receiver.

Figure 2.12.: Lossy channel system example: Alternating bit protocol.

are omitted and only sequence numbers are transmitted. The LCS consists channels c_M and c_A, messages 0 and 1 and the two automata of Fig. 2.12. Channel c_M is used to transmit messages from the sender to the receiver and channel c_A is used to transmit acknowledgements back from the receiver to the sender.

The protocol works as follows. The sender sends a message over channel c_M (transition from s_1 to s_2, actual message omitted) followed by the current sequence number (initially 0). It then waits for an acknowledgement over channel c_A with the same sequence number. During the process of waiting, the sender may repeat sending the message and the sequence number. In an actual implementation of the protocol, the repetition would only happen after some delay. After the acknowledgement has arrived, the procedure is repeated for the next message and the next sequence number (modulo 1).

The receiver receives message and sequence numbers from the message channel c_M (actual message omitted). If the sequence number has the expected value (initially 0), it passes the message off to some handler (transition from r_2 to r_3) and sends the sequence number over the acknowledgement channel c_A and waits for the next message accompanied by the next expected sequence number. The process of sending acknowledgement messages over c_A is repeated until a message with the next

sequence number is received and the procedure is repeated. Messages with non-expected sequence numbers are discarded.

Through the procedure of repeatedly sending messages, sequence numbers and acknowledgement with sequence numbers, the protocol tries to balance the lossiness of the channels to ensure that messages are handled in the correct order on the receiver side. $\Diamond$

2.8. Program Correctness

To formally show the correctness of our main algorithms, we employ Hoare-style proofs in the proof system PW for *partial correctness of while programs* and TW for *total correctness of while programs* of [ABO09].

Definition 2.19 (Syntax of While Programs). While programs are generated by the following grammar:

$$\begin{aligned} S ::= &\ \textbf{skip} \mid u := t \mid S_1;\ S_2 \\ &\mid \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi} \mid \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\,. \end{aligned}$$

Letter u stands for a variable, t for an expression of the same type as variable u, and B stands for a boolean expression. As a shorthand, we define

$$\textbf{if}\ B\ \textbf{then}\ S\ \textbf{fi}\ \equiv\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else skip fi}\,. \qquad \Diamond$$

The **skip** command does a program step without modifying any variables. The rest of the semantics of these simple programs is as expected. See [ABO09, p. 58] for a complete definition.

We express program correctness by so-called *correctness formulas* of the form

$$\{\,p\,\}\ S\,\{\,q\,\}$$

where S is a while program and p and q are assertions. Assertion p is the *precondition* and q is the *postcondition* of the correctness formula. The truth of correctness formulas is defined via the semantics of while programs. For sake of brevity, we paraphrase the definition without having to establish the formal semantics of while programs, which suffices for our purposes of proving algorithms via so-called *proof outlines*.

Definition 2.20 (Correctness). We distinguish partial and total correctness.

(i) We say that the correctness formula $\{ p \}\ S \{ q \}$ is true in the sense of *partial correctness*, and write $\models \{ p \}\ S \{ q \}$, if every *terminating* computation of S that starts in a state satisfying precondition p terminates in a state satisfying postcondition q.

(ii) We say that the correctness formula $\{ p \}\ S \{ q \}$ is true in the sense of *total correctness*, and write $\models_{tot} \{ p \}\ S \{ q \}$, if every computation of S that starts in a state satisfying p *terminates* in a state satisfying precondition p terminates in a state satisfying postcondition q. ◊

The following proof systems and relations between these systems and above definition of correctness give means to formulate detailed correctness proofs of while programs.

Definition 2.21 (Proof System PW). The proof system PW consists of following axioms and rules.
AXIOM 1: SKIP

$$\{ p \}\ \textbf{skip}\ \{ p \}$$

AXIOM 2: ASSIGNMENT

$$\{ p[u := t] \}\ u := t\ \{ p \}$$

where $p[u := t]$ describes the substitution of u in p by t.

RULE 3: COMPOSITION

$$\frac{\{ p \}\ S_1\ \{ r \},\ \{ r \}\ S_2\ \{ q \}}{\{ p \}\ S_1;\ S_2\ \{ q \}}$$

RULE 4: CONDITIONAL

$$\frac{\{ p \wedge B \}\ S_1\ \{ q \},\ \{ p \wedge \neg B \}\ S_2\ \{ q \}}{\{ p \}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{ q \}}$$

RULE 5: LOOP

$$\frac{\{ p \wedge B \}\ S\ \{ p \}}{\{ p \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{ p \wedge \neg B \}}$$

RULE 6: CONSEQUENCE

$$\frac{p \rightarrow p_1,\ \{\,p_1\,\}\ S\ \{\,q_1\,\},\ q_1 \rightarrow q}{\{\,p\,\}\ S\ \{\,q\,\}}$$

◊

In loop rule 5, p is understood as an *invariant* of the loop under consideration.

While proof system PW suffices to show that *if* a program terminates the results meet the specification, we also want to show that programs *do* terminate. Proof system TW for total correctness empowers us to prove program termination.

Definition 2.22 (Proof System TW). The proof system TW consists of axioms and rules 1–4, 6, and the following rule 7.[7]

RULE 7: LOOP II

$$\frac{\begin{array}{l}\{\,p \wedge B\,\}\ S\ \{\,p\,\},\\ \{\,p \wedge B \wedge t = \alpha\,\}\ S\ \{\,t < \alpha\,\},\\ p \Rightarrow t \in W\end{array}}{\{\,p\,\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\,p \wedge \neg B\,\}}$$

where

(i) t is an expression which takes values in an irreflexive partial order $(P, >)$ that is well-founded on the subset $W \subseteq P$,

(ii) α is a simple variable ranging over P and not occurring in p, t, B or S. ◊

When a correctness formula $\{\,p\,\}\ S\,\{\,q\,\}$ is deducible from some set of assumption formulas $\mathcal{A}$ in proof system P, we write

$$\mathcal{A} \vdash_P \{\,p\,\}\ S\,\{\,q\,\}$$

[7] In fact, proof system TW is defined in [ABO09, p. 70], but we need the *while* fragment of the more general notion of *fair total correctness of nondeterministic programs* which exploits well-founded structures for bound functions. To be precise, we use rule 39' of [ABO09, p. 432] for the special case **while** B **do** S **od** $\equiv$ **do** $\square B \rightarrow S$ **od**. We restrict these nondeterministic programs to while programs and dismiss the fairness requirement.

and when the set of assumption $\mathcal{A}$ is empty, we simply write

$$\vdash_P \{p\}\ S\{q\}$$

and call it a *theorem* of proof system P.
Both proof systems PW and TW are shown to be sound in [ABO09, Theorem 3.1, p. 74] and we use this result with following lemma.

Lemma 2.11 (Soundness of PW and TW). Proof systems PW and TW are sound.

(i) Every theorem of proof system PW is correct. Formally,

$$\vdash_{PW} \{p\}\ S\{q\} \quad \text{implies} \quad \models \{p\}\ S\{q\}\ .$$

(ii) Every theorem of proof system TW is correct. Formally,

$$\vdash_{TW} \{p\}\ S\{q\} \quad \text{implies} \quad \models_{tot} \{p\}\ S\{q\}\ .$$

Proof. Shown in [ABO09, Theorem 3.1, p. 74]. □

In practice, we are interested in decoupling proofs for correctness and termination. The following definition introduces the decomposition rule which allows for the combination of two separate correctness and termination proofs.

Definition 2.23 (Decomposition). By the decomposition rule, separate proofs for correctness and termination with a trivial postcondition allow for the deduction of total correctness.
RULE A1: DECOMPOSITION

$$\frac{\begin{array}{l}\vdash_{PW} \{p\}\ S\ \{q\},\\ \vdash_{TW} \{p\}\ S\ \{\mathbf{true}\},\end{array}}{\models_{tot} \{p\}\ S\ \{q\}}$$

◊

For our proofs to be easier to follow, we use *proof outlines* which were first introduced by Owicki and Gries in [OG76]. Roughly speaking, in a proof outline, each line of the underlying program is enclosed in assertions representing pre- and postconditions according to axioms and rules of proof system PW.

Definition 2.24 (Proof Outline for Partial Correctness). Let S^* stand for the program S interspersed with assertions. We define the notion of a *proof outline for partial correctness* inductively by the following formation axioms and rules. A *formation axiom* φ should be read here as a statement: φ is a proof outline (for partial correctness). A *formation rule*

$$\frac{\varphi_1, \ldots, \varphi_k}{\varphi_{k+1}}$$

should be read as a statement: if $\varphi_1, \ldots, \varphi_k$ are proof outlines, then φ_{k+1} is a proof outline.

(i) $\{\, p \,\}$ **skip** $\{\, p \,\}$

(ii) $\{\, p[u := t] \,\}\ u := t\ \{\, p \,\}$

(iii) $$\frac{\{\, p \,\}\ S_1^*\ \{\, r \,\},\ \{\, r \,\}\ S_2^*\ \{\, q \,\}}{\{\, p \,\}\ S_1^*;\ \{\, r \,\}\ S_2^* \{\, q \,\}}$$

(iv) $$\frac{\{\, p \wedge B \,\}\ S_1^*\ \{\, q \,\},\ \{\, p \wedge \neg B \,\}\ S_2^*\ \{\, q \,\}}{\{\, p \,\}\ \textbf{if}\ B\ \textbf{then}\ \{\, p \wedge B \,\}\ S_1^*\ \{\, q \,\}\ \textbf{else}\ \{\, p \wedge \neg B \,\}\ S_2^*\ \{\, q \,\}\ \textbf{fi}\ \{\, q \,\}}$$

(v) $$\frac{\{\, p \wedge B \,\}\ S^*\ \{\, p \,\}}{\{\, \textbf{inv:}\ p \,\}\ \textbf{while}\ B\ \textbf{do}\ \{\, p \wedge B \,\}\ S^*\ \{\, p \,\}\ \textbf{od}\ \{\, p \wedge \neg B \,\}}$$

(vi) $$\frac{p \rightarrow p_1,\ \{\, p_1 \,\}\ S^*\ \{\, q_1 \,\},\ q_1 \rightarrow q}{\{\, p \,\}\ \{\, p_1 \,\}\ S^*\ \{\, q_1 \,\}\ \{\, q \,\}}$$

(vii) $$\frac{\{\, p \,\}\ S^*\ \{\, q \,\}}{\{\, p \,\}\ S^{**}\ \{\, q \,\}}$$
where S^{**} results from S^* by omitting some assertions of the form $\{\, r \,\}$. Thus all assertions of the form $\{\, \textbf{inv:}\ r \,\}$ (and $\{\, \textbf{bd:}\ r \,\}$) remain. ◊

Like proof systems TW and PW, proof outlines for total correctness differ only little from proof outlines for partial correctness.

Definition 2.25 (Proof Outline for Total Correctness). Let S^* stand for a program annotated with assertion, some of them labelled by the keyword **inv**, and integer expressions, all labelled by the keyword **bd**. The notion of a *proof outline for total correctness*[8] is defined as for

[8] For the formation rule which corresponded to the LOOP II rule in [ABO09, p. 83], we have substituted the respective case of rule 39' of [ABO09, p. 432] and varied it to use a single proof outline.

partial correctness (cf. Def. 2.24), except for formation rule (v) dealing with loops, which is replaced by

(viii) $\dfrac{\{\, p \wedge B \,\}\ \alpha := t;\ \{\, p \wedge B \wedge t = \alpha \,\}\ S^*\ \{\, p \wedge t < \alpha \,\},\quad p \Rightarrow t \in W}{\{\,\textbf{inv:}\ p\,\}\{\,\textbf{bd:}\ t\,\}\ \textbf{while}\ B\ \textbf{do}\ \alpha := t;\ \{\, p \wedge B \wedge \alpha = t \,\}\ S^*\ \{\, p \wedge t < \alpha \,\}\ \textbf{od}\ \{\, p \wedge \neg B \,\}}$

where

(i) t is an expression which takes values in an irreflexive partial order $(P, >)$ that is well-founded on the subset $W \subseteq P$,

(ii) α is a simple variable ranging over P and not occurring in p, t, B or S. ♢

The connection between proof outlines and provability with proof systems is established in Theorem 3.2 of [ABO09, p. 81]. We recall it as the following lemma.

Lemma 2.12 (Proof Outlines Imply Correctness Theorems). The existence of a proof outline implies the a correctness theorem in the respective proof system.

(i) Every proof outline for partial correctness implies a theorem of proof system PW. Formally, if $\{\, p \,\}\ S^*\ \{\, q \,\}$ is a proof outline for partial correctness, then

$$\vdash_{PW} \{\, p \,\}\ S\ \{\, q \,\}\ .$$

(ii) Every proof outline for total correctness implies a theorem of proof system TW. Formally, if $\{\, p \,\}\ S^*\ \{\, q \,\}$ is a proof outline for total correctness, then

$$\vdash_{TW} \{\, p \,\}\ S\ \{\, q \,\}\ .$$

Proof. Shown in [ABO09, Theorem 3.2, p. 81]. □

Example 2.11. Consider the simple while program shown in Alg. 2.2, which computes the factorial $n! = \prod_{i=1}^{n} i$ of some natural number n (and by convention $0! = 1$) in a straightforward way. We claim that this pro-

```
1 r := 1;
2 i := 0;
3 while i < n do
4     i := i + 1;
5     r := r · i
6 od
```

Algorithm 2.2: Example algorithm for program correctness: FACTORIAL.

gram satisfies the correctness formula $\{ n \in \mathbb{N} \}$ FACTORIAL $\{ r = n! \}$ in the sense of total correctness, i.e. it terminates for all inputs of $n \in \mathbb{N}$ with the value of variable r being $n!$.

```
1   { n ∈ ℕ }
2  r := 1;
3   { n ∈ ℕ ∧ r = 1 }
4  i := 0;
5   { n ∈ ℕ ∧ r = 1 ∧ i = 0 }
6   { inv: n ∈ ℕ ∧ r = i! }   { bd: n − i }
7  while i ≠ n do
8      α := n − i;
9       { i ≠ n ∧ n ∈ ℕ ∧ r = i! ∧ n − i = α }
10      { n ∈ ℕ ∧ r = i! ∧ n − i = α }
11     i := i + 1;
12      { n ∈ ℕ ∧ r = (i − 1)! ∧ n − (i − 1) = α }
13     r := r · i
14      { n ∈ ℕ ∧ r = (i − 1)! · i ∧ n − (i − 1) = α }
15      { n ∈ ℕ ∧ r = i! ∧ n − i < n − i + 1 = α }
16 od
17  { i = n ∧ n ∈ ℕ ∧ r = i! }
18  { r = n! }
```

Figure 2.13.: Example proof outline for total correctness: FACTORIAL.

To verify this claim, we present proof outline Fig. 2.13 for total correct-

ness which makes use of the assignment axiom and the consequence rule (Def. 2.21). The invariant states that in each iteration, variable r reflects the factorial of the current number of iterations i and that n remains a natural number. It is preserved over the execution of the loop body. Invariant $n \in \mathbb{N} \wedge r = !i$ together with the negated loop condition implies $r = !n$ which is the result we wanted for partial correctness. Bound function $n - i$ expresses that the number of iterations does not exceed the value of n. It takes values in the naturals which are well-founded w.r.t. $<$. Termination then follows from the fact that $n - i$ strictly decreases with each iteration. $\Diamond$

In the following chapter we will put the methodology of employing proof outlines to show correctness to use: We will present separate proof outlines for partial correctness and termination (total correctness with a trivial postcondition) of algorithms, deduce that the corresponding correctness formulas consisting of precondition, algorithm, and postcondition are theorems by Lemma 2.12 (proof outlines are theorems), and combine the two correctness formulas for partial correctness and termination via Def. 2.23 (decomposition) to show that the partial correctness formula is true even in the sense of total correctness as described in Def. 2.20, i.e. every computation that starts in a state that satisfies the precondition terminates in a state that satisfies the postcondition.

CHAPTER THREE

Algorithmic Framework

Technological progress has [. . .] provided us with more efficient means for going backwards.

— Aldous Huxley, Writer

Contents

3.1 Proof of the Basic BR 52
 3.1.1 Partial Correctness 53
 3.1.2 Termination 63
 3.1.3 Total Correctness 68
3.2 Extending the Basic BR 69
3.3 Framework 79
3.4 Search Space Constructions 81
 3.4.1 Optimized Predecessors and Distance Reduction 82
 3.4.2 Witness Traces and Search Strategies 84
3.5 Arguments for Termination and Correctness 85

The decision procedure from [AČJT96, FS01] performs a fixed-point iteration to compute the basis of states that are backward reachable from $\uparrow F$

(cf. Sect. 2.5 on p. 30). In this chapter, we provide an axiomatic Hoare-style proof for the basic backward reachability analysis (basic BR) and introduce our algorithmic framework as a refinement of the basic backward reachability analysis. We discuss general means to apply optimizations and refrain from computing a basis of the complete set of backward reachable states if a transition sequence to prove coverability is found. In Sect. 3.5 we give high-level arguments for termination and correctness, whereas a detailed proof follows in Ch. 4 on p. 87.

Users of our algorithmic framework benefit in different ways: on the one hand, they may get to a decision procedure for some new system class that is a WSTS with less effort and on the other hand, proofs for optimizations—possibly specially crafted for their class—may become more elegant, as we will discuss in Ch. 5 on p. 130.

Beginning this chapter, we formulate a new axiomatic Hoare-style proof for the basic backward reachability analysis that we base our framework on.

3.1. Proof of the Basic BR

There exist correctness and termination proofs for an abstract set-saturation method, that iteratively computes $pre^*(\uparrow X) = \bigcup_{i=0}^{\infty} pre^i(\uparrow F)$ (cf. [FS01, AČJT96]), which basically use some form of the property that $pre^*(\uparrow X)$ can be computed in a finite number of steps as stated in Lemma 2.10 on p. 27. While such high-level proofs are beautifully concise, we strive for a proof in close relation to program code.

Moreover, the basic backward reachability analysis, in the form we stated in the previous chapter, works differently than the abstract set-saturation methods. In each iteration it selects a single element of W to compute a pred-basis for, as opposed to compute the pred-basis of the whole set W. The basic backward reachability analysis in the previous chapter is closer to the algorithm of [AJKP98], where a proof by induction on the length of some transition sequence from the initial states to the UCS of final states is employed to show correctness.

We believe that, despite the size of our axiomatic proof of the relatively small algorithm, the benefits of understanding the basic BR and its properties more thoroughly are worthwhile.

The proof consists of three parts. We begin by showing that if the algorithm terminates, its result is an answer to the coverability problem described by the input. After establishing that the algorithm terminates for every valid input, we combine both proofs to conclude total correctness.

3.1.1. Partial Correctness

To create an axiomatic Hoare-style proof for partial of the algorithm, we need to establish the following lemma. It reveals a nice property of the predecessors $pre^*(\uparrow X)$ of some upward-closed set. In detail, it allows the introduction of a $\uparrow$-operator at a certain place where it is not obvious. While $pre(\uparrow X) \subseteq \uparrow pre(\uparrow X)$ holds by the definition of $\uparrow$, the inverse inclusion is not true in general: With upward-compatibility, several transition steps may be performed on the right-hand side for every single step on the left-hand side (cf. Fig. 2.4 on p. 20).

Lemma 3.1 (Expansion of pre^* and the Upward Closure). Given a WSTS $(S, \rightarrow, \preceq)$ and a set $X \subseteq S$, for the expansion of $pre^*(\uparrow X) = \uparrow X \cup pre^*(pre(\uparrow X))$ (cf. Lemma 2.7-4 on p. 25) the following equality holds:

$$\uparrow X \cup \uparrow pre^*(pre(\uparrow X)) = \uparrow X \cup \uparrow pre^*(\uparrow pre(\uparrow X))$$

Proof. Let $(S, \rightarrow, \preceq)$ be a WSTS and $X \subseteq S$ a set of states.
We show the inclusion $\uparrow X \cup \uparrow pre^*(pre(\uparrow X)) \subseteq \uparrow X \cup \uparrow pre^*(\uparrow pre(\uparrow X))$: This holds by definition of upward-closure and monotonicity of pre^* (cf. Def. 2.7 on p. 20, Lemma 2.7-2 on p. 24).
To show the inverse inclusion,

$$\uparrow X \cup \uparrow pre^*(\uparrow pre(\uparrow X)) \subseteq \uparrow X \cup \uparrow pre^*(pre(\uparrow X))\,,$$

we distinguish the following two cases. Let $y \in \uparrow X \cup \uparrow pre^*(\uparrow pre(\uparrow X))$.

- In case $y \in \uparrow X$, it follows immediately that state y is in $\uparrow X \cup \uparrow pre^*(pre(\uparrow X))$.
- In case $y \notin \uparrow X$, we intend to prove $\uparrow pre^*(\uparrow pre(\uparrow X)) \subseteq \uparrow pre^*(\uparrow X)$ in order to use $\uparrow X \cup \uparrow pre^*(pre(\uparrow X)) = \uparrow pre^*(\uparrow X)$ by the expansion of pre^* and the distributivity of $\uparrow$ over $\cup$ (cf. Lemma 2.7-4 on p. 25, Lemma 2.5-2 on p. 21).

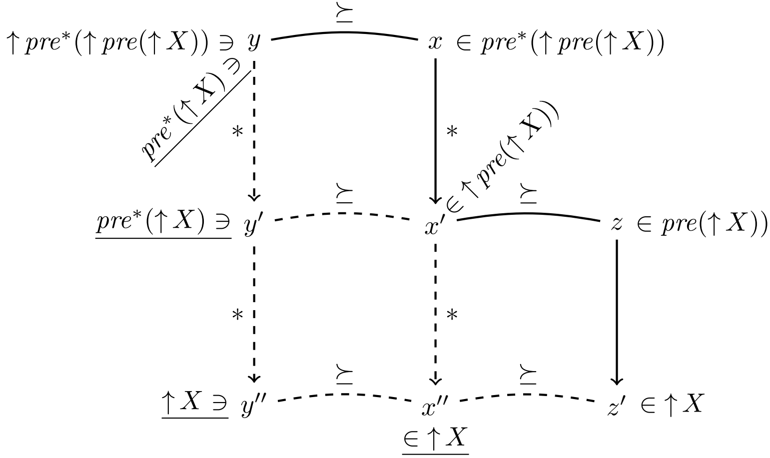


Figure 3.1.: Upward-compatibility in the proof of Lemma 3.1.

Let $y \in \uparrow pre^*(\uparrow pre(\uparrow X))$. By the definition of predecessors and the upward-closure, there exist states $x \in pre^*(\uparrow pre(\uparrow X))$, $x' \in \uparrow pre(\uparrow X)$, $z \in pre(\uparrow X)$ and $z' \in \uparrow X$ with

$$y \succeq x \to^* x' \succeq z \to z' \in \uparrow X\,.$$

The upward-compatibility in turn ensures the existence of states $x'' \succeq z'$, $y' \succeq x'$, and $y'' \succeq x''$ s.t. $y \to^* y' \to^* y'' \succeq x'' \succeq z' \in \uparrow X$ which expresses that y is a predecessor of a state in $\uparrow X$, i.e. $y \in pre^*(\uparrow X)$.

A graphical representation is given in Fig. 3.1. Dashed lines represent guarantees by the upward-compatibility of the WQO with $\to$ and underlined sets indicate deductions via upward-compatibility.

Hence, by Lemma 2.8 on p. 26, $\uparrow pre^*(\uparrow pre(\uparrow X)) \subseteq pre^*(\uparrow X) = \uparrow pre^*(\uparrow X) = \uparrow X \cup \uparrow pre^*(pre(\uparrow X))$.

We conclude that $\uparrow X \cup \uparrow pre^*(pre(\uparrow X)) = \uparrow X \cup \uparrow pre^*(\uparrow pre(\uparrow X))$ holds. $\square$

To formally show the correctness of Alg. 2.1 on p. 30 we give a proof outline for partial correctness in Fig. 3.2 on p. 62. There, we will use the following abbreviations:

$$Inv_{VW}(V,W) := pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \tag{3.1}$$

$$Inv_F(V,W) := |V| \in \mathbb{N} \wedge \uparrow V \cup pre^*(\uparrow W) = pre^*(\uparrow F) \tag{3.2}$$

To begin with, we show a useful property of the first invariant which states that if the one-step predecessors of $\uparrow V$ are contained in the union of $\uparrow V$ and $\uparrow W$, then this inclusion can be lifted to $pre^*(\uparrow V) \subseteq \uparrow V \cup pre^*(\uparrow W)$. Notice that, on the right-hand side of the inclusion, pre^* is applied only on $\uparrow W$. The intuition for both inclusions is that if $\uparrow W$ is empty, $\uparrow V$ is transitively closed w.r.t. *pre*. If $\uparrow V$ is not closed in this sense, then $\uparrow W$ contains all the states which $\uparrow V$ is missing to be transitively closed w.r.t. *pre*. This relationship is lifted to pre^*.

Lemma 3.2. If $(S, \rightarrow, \preceq)$ is a WSTS and V, W are subsets of S, then

$$pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \quad \text{implies} \quad pre^*(\uparrow V) \subseteq \uparrow V \cup pre^*(\uparrow W)\,.$$

Proof. Let $(S, \rightarrow, \preceq)$ be a WSTS and V, W subsets of S. By induction we prove that the implication

$$pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \quad \text{implies} \quad \bigcup_{i=0}^{n} pre^i(\uparrow V) \subseteq \uparrow V \cup \bigcup_{i=0}^{n} pre^i(\uparrow W)$$

holds for any natural n.

Base case. Let $n = 0$. The implication obviously holds as the upward-closed set $\bigcup_{i=0}^{n} pre^i(\uparrow V) = \uparrow V$ is a subset of $\uparrow V \cup \bigcup_{i=0}^{n} pre^i(\uparrow W) = \uparrow V \cup \uparrow W$.

Induction hypothesis. Let the implication be proved for n.

Inductive step. We show that under the assumption of the induction hypothesis, the statement is also true for $n+1$.

Consider some state $x \in \bigcup_{i=0}^{n+1} pre^i(\uparrow V)$. If state x is a member of some set $pre^i(\uparrow V)$ for $i \leq n$, then, by application of the induction hypothesis, it is also a member of $\uparrow V \cup \bigcup_{i=0}^{n} pre^i(\uparrow W)$ which is a subset of $\uparrow V \cup \bigcup_{i=0}^{n+1} pre^i(\uparrow W)$.

We assume x is in set $pre^{n+1}(\uparrow V)$. By the definition of the predecessors (Def. 2.10 on p. 24) there exists a state $x' \in pre^n(\uparrow V)$ to which x is a predecessor, i.e. $x \rightarrow x'$. From the induction hypothesis we know that $pre^n(\uparrow V) \subseteq \bigcup_{i=0}^{n} pre^i(\uparrow V)$ is a subset of $\uparrow V \cup \bigcup_{i=0}^{n} pre^i(\uparrow W)$. Therefore, state x is a member of

$$pre(\uparrow V \cup \bigcup_{i=0}^{n} pre^i(\uparrow W))$$

which can be written with the use of distributivity (Lemma 2.7-1 on p. 24) as $pre(\uparrow V) \cup \bigcup_{i=0}^{n} pre(pre^i(\uparrow W))$.

From the implication's prerequisite, $pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W$, we deduce

$$pre(\uparrow V) \cup \bigcup_{i=0}^{n} pre(pre^i(\uparrow W)) \quad \subseteq \quad \uparrow V \cup \uparrow W \cup \bigcup_{i=0}^{n} pre(pre^i(\uparrow W))$$

which we rearrange to

$$pre(\uparrow V) \cup \bigcup_{i=0}^{n} pre(pre^i(\uparrow W)) \quad \subseteq \quad \uparrow V \cup \bigcup_{i=0}^{n+1} pre^i(\uparrow W)\,.$$

Therefore, x is in $\uparrow V \cup \bigcup_{i=0}^{n+1} pre^i(\uparrow W)$ and we conclude that the following subset relation holds.

$$\bigcup_{i=0}^{n+1} pre^i(\uparrow V) \quad \subseteq \quad \uparrow V \cup \bigcup_{i=0}^{n+1} pre^i(\uparrow W)$$

By the principle of induction, the statement

$$pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \quad \text{implies} \quad \bigcup_{i=0}^{n} pre^i(\uparrow V) \subseteq \uparrow V \cup \bigcup_{i=0}^{n} pre^i(\uparrow W) \tag{3.3}$$

holds for any natural n.

As we know from the stabilization of predecessors of upward-closed sets

(Lemma 2.10 on p. 27), there exist natural numbers k_1 and k_2, s.t. for any $j \in \mathbb{N}$,

$$\uparrow \bigcup_{i=0}^{k_1+j} pre^i(\uparrow V) = pre^*(\uparrow V) \quad \text{and} \quad \uparrow \bigcup_{i=0}^{k_2+j} pre^i(\uparrow W) = pre^*(\uparrow W)\,.$$

Thus, we are allowed to set $n = \max\{\,k_1, k_2\,\}$ and substitute the unions from 0 to n in Eq. 3.3 by pre^*-operations and conclude our proof: Statement

$$pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \quad \text{implies} \quad pre^*(\uparrow V) \subseteq \uparrow V \cup pre^*(\uparrow W)$$

holds. □

Most steps in the proof outline Fig. 3.2 on p. 62 work mechanically by forward elimination of conjunctions by the consequence rule and backward substitution by the assignment axiom (cf. Def. 2.21 on p. 44 or rather Def. 2.24 on p. 47). For the implication between lines 15 and 17, we employ Lemma 3.3 and Lemma 3.4.

The first of these lemmas establishes the fact that the relationship between sets V and W remains intact over a single loop iteration. The lemma is used for the case that the test in line 6 of Alg. 2.1 on p. 30, **if** $x \notin \uparrow V$ **then**, is positive and the body of the **if**-construct is executed and the predecessors of x are added to W. The condition $x \notin \uparrow V$ is not needed for the proof.

Lemma 3.3. Let $(S, \rightarrow, \preceq)$ be a WSTS, $x \in S$ a state and $V, W \subseteq S$ finite sets of states. The following implication holds.

$$Inv_{VW}(V, W \cup \{\,x\,\}) \quad \Rightarrow \quad Inv_{VW}(V \cup \{\,x\,\}, W \cup pb(x))$$

Proof. The implication without abbreviation by Eq. 3.1 is

$$\begin{aligned} & pre(\uparrow V) \subseteq \uparrow V \cup \uparrow(W \cup \{\,x\,\}) \\ \Rightarrow \quad & pre(\uparrow(V \cup \{\,x\,\})) \subseteq \uparrow(V \cup \{\,x\,\}) \cup \uparrow(W \cup pb(x))\,. \end{aligned}$$

Using distributivity of $\uparrow$ and *pre* over $\cup$ (Lemma 2.5-2 on p. 21, Lemma 2.7-1 on p. 24) and the definition of the pred-basis (Def. 2.11 on p. 29), we have to prove the following implication.

$$\begin{aligned} & pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \cup \uparrow x \\ \Rightarrow \quad & pre(\uparrow V) \cup pre(\uparrow x) \subseteq \uparrow V \cup \uparrow x \cup \uparrow W \cup \uparrow \text{basis}(pre(\uparrow x)) \,. \end{aligned}$$

Assume the implication's left-hand side holds. We turn to the right-hand side and use commutativity of $\cup$ and apply the definition of a basis (Def. 2.7 on p. 20) which leaves us with the following condition to prove:

$$pre(\uparrow V) \cup pre(\uparrow x) \quad \subseteq \quad \uparrow V \cup \uparrow W \cup \uparrow x \cup \uparrow pre(\uparrow x) \tag{3.4}$$

We know $pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W \cup \uparrow x$ from the left-hand side of the implication and obviously $pre(\uparrow x) \subseteq \uparrow pre(\uparrow x)$ holds. Therefore, the above inclusion is implied by $pre(\uparrow V) \subseteq \uparrow V \cup \uparrow(W \cup \{ x \})$ and we conclude that the implication holds. □

The second lemma guarantees that the algorithm's result V is in fact a basis of the states backward reachable from $\uparrow F$, even when the algorithm performs the step from x to $pb(x)$.

Lemma 3.4. Let $(S, \rightarrow, \preceq)$ be a WSTS, $x \in S$ a state and $V, W, F \subseteq S$ finite sets of states.

$$Inv_F(V, W \cup \{ x \}) \quad \Rightarrow \quad Inv_F(V \cup \{ x \}, W \cup pb(x))$$

Proof. The fact that V is finite implies that $V \cup \{ x \}$ is finite. By abbreviation Eq. 3.2, we remain to show equality of following two sets.

$$\uparrow V \cup pre^*(\uparrow(W \cup \{ x \})) \quad = \quad \uparrow(V \cup \{ x \}) \cup pre^*(\uparrow(W \cup pb(x)))$$

We apply the lemmas of the previous section on upward-closure and the predecessor functions to show a direct transformation from one set to the other. Let $(S, \rightarrow, \preceq)$ be a WSTS, $x \in S$ a state and $V, W, F \subseteq S$ finite sets of states. For convenience, we underlined the parts of each line that

changed w.r.t. the previous.

$$
\begin{aligned}
& \uparrow V \cup pre^*(\uparrow(W \cup \{\, x \,\})) \\
= \; & \uparrow V \cup pre^*(\underline{\uparrow W \cup \uparrow x}) && \text{(Distr. } \uparrow, \cup \text{ [Lemma 2.5-2 on p. 21])} \\
= \; & \uparrow V \cup \underline{pre^*(\uparrow W) \cup pre^*(\uparrow x)} && \text{(Distr. } pre, \cup \text{ [Lemma 2.7-1 on p. 24])} \\
= \; & \uparrow V \cup pre^*(\uparrow W) \cup \underline{\uparrow pre^*(\uparrow x)} \\
& && (pre^* \text{ of a UCS [Lemma 2.8 on p. 26])} \\
= \; & \uparrow V \cup pre^*(\uparrow W) \cup \underline{\uparrow(\uparrow x \cup pre^*(pre(\uparrow x)))} \\
& && \text{(Expan. } pre^* \text{ [Lemma 2.7-4 on p. 25])} \\
= \; & \uparrow V \cup pre^*(\uparrow W) \cup \underline{\uparrow\uparrow x \cup \uparrow pre^*(pre(\uparrow x))} \\
& && \text{(Distr. } \uparrow, \cup \text{ [Lemma 2.5-2 on p. 21])} \\
= \; & \uparrow V \cup pre^*(\uparrow W) \cup \underline{\uparrow x} \cup \uparrow pre^*(pre(\uparrow x)) \\
& && \text{(Idem. } \uparrow \text{ [Lemma 2.5-1 on p. 21])} \\
= \; & \uparrow V \cup pre^*(\uparrow W) \cup \underline{\uparrow x \cup \uparrow pre^*(\uparrow pre(\uparrow x))} && \text{(Lemma 3.1)} \\
= \; & \uparrow V \cup pre^*(\uparrow W) \cup \uparrow x \cup \underline{pre^*(\uparrow pre(\uparrow x))} \\
& && (pre^* \text{ of a UCS [Lemma 2.8 on p. 26])} \\
= \; & \uparrow V \cup \underline{\uparrow x \cup pre^*(\uparrow W)} \cup pre^*(\uparrow pre(\uparrow x)) && \text{(Comm. } \cup) \\
= \; & \uparrow V \cup \uparrow x \cup \underline{pre^*(\uparrow W \cup \uparrow pre(\uparrow x))} \\
& && \text{(Distr. } pre^*, \cup \text{ [Lemma 2.7-1 on p. 24])} \\
= \; & \uparrow V \cup \uparrow x \cup pre^*(\uparrow W \cup \underline{\uparrow \mathrm{basis}(pre(\uparrow x))}) \\
& && (\uparrow \mathrm{basis}(X) = \uparrow X \text{ [Def. 2.7 on p. 20])} \\
= \; & \uparrow V \cup \uparrow x \cup pre^*(\uparrow W \cup \uparrow \underline{pb(x)}) && \text{(pred-basis [Def. 2.11 on p. 29])} \\
= \; & \underline{\uparrow(V \cup \{\, x \,\})} \cup pre^*(\underline{\uparrow(W \cup pb(x))}) \\
& && \text{(Distr. } \uparrow, \cup \text{ [Lemma 2.5-2 on p. 21])}
\end{aligned}
$$

Thus, the two sets are equal and the implication between $Inv_F(V, W \cup \{\, x \,\})$ and $Inv_F(V \cup \{\, x \,\}, W \cup pb(x))$ holds. □

The next lemma ensures that the relationship between sets V and W remains intact over a single loop iteration in case that x, the state selected and removed from W, is contained in the upward-closure of V. We have to

explain, why it is admissible to *not* add the basis of one-step predecessors of x, $pb(x)$, to W.

Lemma 3.5. Let $(S, \to, \preceq)$ be a WSTS, $x \in S$ a state and $V, W, F \subseteq S$ finite sets of states. Then the following implication is correct.

$$\begin{aligned} & x \in \uparrow V \wedge \mathit{Inv}_{VW}(V, W \cup \{\, x \,\}) \wedge \mathit{Inv}_F(V, W \cup \{\, x \,\}) \\ \Rightarrow \quad & \mathit{Inv}_{VW}(V, W) \wedge \mathit{Inv}_F(V, W) \end{aligned}$$

Proof. Expand abbreviations Eq. 3.1 and Eq. 3.2 to

$$\begin{aligned} & x \in \uparrow V \wedge \mathit{pre}(\uparrow V) \subseteq \uparrow V \cup \uparrow(W \cup \{\, x \,\}) \\ & \wedge |V| \in \mathbb{N} \wedge \uparrow V \cup \mathit{pre}^*(\uparrow(W \cup \{\, x \,\})) = \mathit{pre}^*(\uparrow F) \\ \Rightarrow \quad & \mathit{pre}(\uparrow V) \subseteq \uparrow V \cup \uparrow W \\ & \wedge |V| \in \mathbb{N} \wedge \uparrow V \cup \mathit{pre}^*(\uparrow W) = \mathit{pre}^*(\uparrow F)\,. \end{aligned}$$

Assume the left-hand side of the implication holds and show that each conjunct of the right-hand side follows.

1. Show that $\mathit{pre}(\uparrow V) \subseteq \uparrow V \cup \uparrow W$ is implied. By distributivity of $\uparrow$ (Lemma 2.5-2 on p. 21), we know that $\uparrow(W \cup \{\, x \,\})$ is the same as $\uparrow W \cup \uparrow x$. As the implication's left-hand side tells us that x is a member of $\uparrow V$ which implies $\uparrow x \subseteq \uparrow V$, we can collapse the inclusion on the left-hand side, $\mathit{pre}(\uparrow V) \subseteq \uparrow V \cup \uparrow W \cup \uparrow x$, to $\mathit{pre}(\uparrow V) \subseteq \uparrow V \cup \uparrow W$ and conclude this part of the proof.

2. Show that $|V| \in \mathbb{N}$ is implied. This is preserved as it already holds on the left-hand side.

3. Show that $\uparrow V \cup \mathit{pre}^*(\uparrow W) = \mathit{pre}^*(\uparrow F)$ is implied. We rather show the equality of $\uparrow V \cup \mathit{pre}^*(\uparrow W)$ and $\uparrow V \cup \mathit{pre}^*(\uparrow(W \cup \{\, x \,\}))$, as we know from the premise that the latter set is the same as $\mathit{pre}^*(\uparrow F)$.

   a) Show $\uparrow V \cup \mathit{pre}^*(\uparrow W) \subseteq \uparrow V \cup \mathit{pre}^*(\uparrow(W \cup \{\, x \,\}))$: This direction is rather obvious as the inclusion

   $$\begin{aligned} \uparrow V \cup \mathit{pre}^*(\uparrow W) &\subseteq \uparrow V \cup \mathit{pre}^*(\uparrow W) \cup \mathit{pre}^*(\uparrow x) \\ &= \uparrow V \cup \mathit{pre}^*(\uparrow(W \cup \{\, x \,\})) \end{aligned}$$

clearly holds as application of distributivity of $\uparrow$ and pre^* (Lemma 2.5-2 on p. 21, Lemma 2.7-1 on p. 24) yields that $pre^*(\uparrow(W \cup \{ x \}))$ is the same as $pre^*(\uparrow W) \cup pre^*(\uparrow x)$.

b) Show $\uparrow V \cup pre^*(\uparrow(W \cup \{ x \})) \subseteq \uparrow V \cup pre^*(\uparrow W)$: Again, we use that $pre^*(\uparrow(W \cup \{ x \})) = pre^*(\uparrow W) \cup pre^*(\uparrow x)$, as well as $\uparrow V$ being a subset of $\uparrow V \cup pre^*(\uparrow W)$. Therefore, we can further reduce both sides of the inclusion we are to prove, and it remains to show

$$pre^*(\uparrow x) \subseteq \uparrow V \cup pre^*(\uparrow W)\,.$$

As $x \in \uparrow V$ and therefore, $\uparrow x \subseteq \uparrow V$, we use monotonicity of the pre^* (Lemma 2.7-2 on p. 24), to show

$$pre^*(\uparrow V) \subseteq \uparrow V \cup pre^*(\uparrow W)$$

instead. In fact, this inclusion was already shown to hold in Lemma 3.2 under the assumption $pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W$, which holds by item 1.

In summary, we have shown that if the implication's left-hand side is true, then sets $\uparrow V \cup pre^*(\uparrow(W \cup \{ x \}))$ and $\uparrow V \cup pre^*(\uparrow W)$ are equal and that $pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W$ indeed follows from the implication's left-hand side.

This closes the proof of the implication. □

The characteristics of the selection function are comprised in the following lemma.

Lemma 3.6. Given a set W, the following axiom is true in the sense of partial and total correctness.

AXIOM: SELECT

$$\{ W \neq \emptyset \}\ x := select(W)\ \{ x \in W \}$$

Proof. The definition of the selection function (Def. 2.13 on p. 30) states that *select* satisfies $W \neq \emptyset \Rightarrow select(W) \in W$ for any set W. □

// $Inv_{VW}(V, W) := pre(\uparrow V) \subseteq \uparrow V \cup \uparrow W$
// $Inv_F(V, W) := |V| \in \mathbb{N} \land \uparrow V \cup pre^*(\uparrow W) = pre^*(\uparrow F)$

1 $\{\, |F| \in \mathbb{N} \,\}$ `// Needed only in the termination proof.`
2 $W := F;$
3 $\{\, W = F \,\}$
4 $V := \emptyset;$
5 $\{\, V = \emptyset \land W = F \land Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
6 $\{$ **inv:** $Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
7 **while** $W \neq \emptyset$ **do**
8 $\{\, W \neq \emptyset \land Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
9 $x := select(W);$
10 $\{\, x \in W \land Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
11 $W := W \setminus \{\, x \,\};$
12 $\{\, Inv_{VW}(V, W \cup \{\, x \,\}) \land Inv_F(V, W \cup \{\, x \,\}) \,\}$
13 **if** $x \notin \uparrow V$ **then**
14 $\{\, x \notin \uparrow V$
15 $\land\, Inv_{VW}(V, W \cup \{\, x \,\}) \land Inv_F(V, W \cup \{\, x \,\}) \,\}$
`/* See Lemma 3.3 and Lemma 3.4. */`
16 $\{\, Inv_{VW}(V \cup \{\, x \,\}, W \cup pb(x))$
17 $\land\, Inv_F(V \cup \{\, x \,\}, W \cup pb(x)) \,\}$
18 $V := V \cup \{\, x \,\};$
19 $\{\, Inv_{VW}(V, W \cup pb(x)) \land Inv_F(V, W \cup pb(x)) \,\}$
20 $W := W \cup pb(x)$
21 $\{\, Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
22 **else**
23 $\{\, x \in \uparrow V$
24 $\land\, Inv_{VW}(V, W \cup \{\, x \,\}) \land Inv_F(V, W \cup \{\, x \,\}) \,\}$
`/* See Lemma 3.5. */`
25 $\{\, Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
26 **skip**
27 $\{\, Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
28 **fi**
29 $\{\, Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
30 **od**
31 $\{\, W = \emptyset \land Inv_{VW}(V, W) \land Inv_F(V, W) \,\}$
32 $\{\, |V| \in \mathbb{N} \land \uparrow V = pre^*(\uparrow F) \,\}$

Figure 3.2.: Proof outline for partial correctness of Alg. 2.1 on p. 30.

We are now fit to construct a proof outline for partial correctness of the basic BR. The proof outline given in Fig. 3.2 allows us to state Proposition 3.7. At the top of the figure, the abbreviations $Inv_{VW}(V, W)$ and $Inv_F(V, W)$ are listed. As mentioned before, most steps in the proof outline work mechanically by forward elimination of conjunctions and backward substitution. The effect of statement $x := select(W)$ is defined in Lemma 3.6. For partial correctness, precondition **true** would suffice, but as we intent to combine the partial correctness proof with a termination proof to gain a total correctness proof by decomposition (Def. 2.23 on p. 46), we have to anticipate a precondition that matches the one for termination. Thus, the precondition demands the set of final states F to be finite, as is needed for termination.

Proposition 3.7. Consider $(S, \rightarrow, \preceq)$ a WSTS with decidable $\preceq$ and effective pred-basis. If the basic BR is applied on a finite set $F \subseteq S$ and it terminates, its result V is finite and $\uparrow V = pre^*(\uparrow F)$ holds. Formally,

$$\vdash_{PW} \{\, |F| \in \mathbb{N} \,\} \text{ Basic BR } \{\, |V| \in \mathbb{N} \wedge \uparrow V = pre^*(\uparrow F) \,\} \;.$$

Proof. By the proof outline in Fig. 3.2 and Lemma 2.12 on p. 48 (proof outlines imply correctness theorems). □

3.1.2. Termination

To answer the question whether the basic backward reachability analysis terminates, we have to construct a bound expression t which takes values in an irreflexive partial order and which decreases with each loop iteration. For the expression t and the irreflexive partial order, we choose $t = (V, W) \in S \times S$ and the following lexicographical order.

Definition 3.1 (Lexicographical Order for Termination of the Basic BR). Given a QO $\preceq$, we define the lexicographical order $>^{bbr}_{lex}$ on pairs of finite sets s.t. the upward-closures of the first components are in proper subset relation or they are equal and the second components are in proper subset relation. Formally, for pairs of finite sets (V, W) and (V', W'), we define

$$(V, W) >^{bbr}_{lex} (V', W') :\Leftrightarrow \uparrow V \subset \uparrow V' \vee (\uparrow V = \uparrow V' \wedge W \supset W') \,. \quad \Diamond$$

While it may seem contra-intuitive that $(V, W) >^{bbr}_{lex} (V', W')$ if $\uparrow V'$ is larger than $\uparrow V$ (and thus $V \subset V'$), this property is exactly what is needed to keep the axiomatic Hoare-style proof for termination simple. It spares us from computing a precise integer bound on the number of loop iterations.

Note that the subset relation $\subseteq$ is a partial order and $\subset$ is an irreflexive partial order. The argument we want to make is that $t = (V, W)$ decreases with each iteration of Alg. 2.1 on p. 30 w.r.t. $>^{bbr}_{lex}$ and that $>^{bbr}_{lex}$ is well-founded on pairs of finite sets, i.e. there are no infinite decreasing sequences, as required for total correctness of loops (Def. 2.25 on p. 47). We formalize this fact in the following lemma.

Lemma 3.8 (Well-Foundedness of $>^{bbr}_{lex}$). The lexicographical order for termination of Alg. 2.1 on p. 30, $>^{bbr}_{lex}$, is well-founded on pairs of finite sets.

Proof. We prove the well-foundedness of $>^{bbr}_{lex}$ on pairs of finite sets in two steps:

1. The proper superset relation $\supset$ is well-founded on finite sets as there exists no infinite strictly decreasing sequence $X_1 \supset X_2 \supset \cdots$ of finite sets $X_1, X_2, \ldots$.

2. From Lemma 2.9 on p. 27 (stabilization) we know that every ascending chain of upward-closed sets eventually stabilizes, or, phrased differently, there exists no strictly increasing sequence $\uparrow X_1 \subset \uparrow X_2 \subset \cdots$ of upward-closed sets $\uparrow X_1, \uparrow X_2, \ldots$ (cf. proof of Lemma 2.9 on p. 27). Hence, the subset relation is well-founded on upward-closed sets.

   The lexicographical order $>^{bbr}_{lex}$ uses the subset relation on the upward-closure operator of the first component of the pair of finite sets.

As both relations used by $>^{bbr}_{lex}$ are well-founded on the sets they are applied to, the lexicographical order itself is well-founded on pairs of finite sets.

This concludes the proof. □

With the lexicographical order at hand, we can turn to the proof outline for total correctness in Fig. 3.3 and summarize the termination of the algorithm in a lemma following below. But first, we take a detailed look at the outline. Again, the effect of statement $x := \mathit{select}(W)$ is defined in Lemma 3.6 and most steps in the proof outline work mechanically by forward elimination of conjunctions and backward substitution. We use the abbreviation

$$\mathit{Inv}_{\mathit{fin}} := |V| \in \mathbb{N} \wedge |W| \in \mathbb{N}$$

to shorten the assertions. The precondition states that the set of final states F has to be finite. It follows, that both sets $V\ (= \emptyset)$ and $W\ (= F)$ are finite at loop entry. Finiteness of these two sets is the invariant for our termination proof. It is preserved as the union-operations are applied on finite sets only—note that $pb(x)$ is finite (cf. Def. 2.11 on p. 29). The pair (V, W) is the term whose value decreases with each iteration w.r.t. the particular lexicographical order $<_{lex}^{bbr}$, which is just the inverse of $>_{lex}^{bbr}$. The first assertion within the loop body contains the conjunct $(V, W) = \alpha$, where α is a variable that is used to store the current value of (V, W) in order to compare the values of (V, W) at the beginning and at the end of each iteration. Over the statements in the loop body, the relation between α and the pair (V, W), which changes by the various assignments, is kept visible by application of backward substitution.
We explore the main implication step in the proof outline for termination of the algorithm in the following lemma.

Lemma 3.9. The implication between line 20 and line 22 in the proof outline of Fig. 3.3 for termination of the basic BR holds.

Proof. Assume the assertion in line 20,

$$x \in V \wedge x \notin \uparrow(V \setminus \{\, x \,\}) \wedge \mathit{Inv}_{\mathit{fin}} \wedge (V \setminus \{\, x \,\}, (W \cup \{\, x \,\}) \setminus pb(x)) = \alpha\,,$$

holds. We need to show that

$$\mathit{Inv}_{\mathit{fin}} \wedge (V, W) <_{lex}^{bbr} \alpha$$

follows.
As the satisfaction of $\mathit{Inv}_{\mathit{fin}}$ is preserved, we concentrate on $(V, W) <_{lex}^{bbr} \alpha$. Here, we compare the values of V and W at the beginning of the loop, α, with the updated sets (V, W).

The question is: "Does $(V,W) <^{bbr}_{lex} (V \setminus \{x\}, (W \cup \{x\}) \setminus pb(x))$ hold?"

Observe, that for any set X and any state y, it holds that if $y \notin \uparrow X$ it follows that $\uparrow(X \cup \{y\}) \supset \uparrow X$ as y is in $\uparrow(X \cup \{y\})$ by the definition of the upward-closure operator (Def. 2.7 on p. 20).

From $x \notin \uparrow(V \setminus \{x\})$ and the above statement $\uparrow V \supset \uparrow(V \setminus \{x\})$ follows if $x \in V$. The assumption already states $x \in V$ and we can conclude that

$$\uparrow V \supset \uparrow(V \setminus \{x\})$$

holds indeed.

By definition of the order $<^{bbr}_{lex}$ (Def. 3.1), the fact that $\uparrow V$ is a proper superset of $\uparrow(V \setminus \{x\})$ suffices for the question to be answered positively. The condition

$$(V,W) <^{bbr}_{lex} (V \setminus \{x\}, (W \cup \{x\}) \setminus pb(x))$$

is satisfied and the implication holds.

This concludes the proof. □

The second implication that does not stem from mechanical backward substitution or elimination of conjuncts is between line 24 and line 25. We prove the implication in the following lemma.

Lemma 3.10. The implication between line 24 and line 25 in the proof outline of Fig. 3.3 for termination of the basic BR holds.

Proof. We need to show the implication

$$x \in \uparrow V \wedge x \notin W \wedge Inv_{fin} \wedge (V, W \cup \{x\}) = \alpha \quad \Rightarrow \quad Inv_{fin} \wedge (V,W) <^{bbr}_{lex} \alpha\,.$$

We assume that the implication's left-hand side holds and show that the right-hand side follows.

Obviously, conjunct Inv_{fin} holds. As intent to compare $\alpha = (V, W \cup \{x\})$ with (V,W), we observe that V is unchanged. By Def. 3.1 of $<^{bbr}_{lex}$ we have to prove that $W \subset W \cup \{x\}$ holds. This simply follows from the assumption $x \notin W$.

Therefore we conclude $(V,W) <^{bbr}_{lex} \alpha$ and the implication holds. □

```
    // Inv_fin := |V| ∈ ℕ ∧ |W| ∈ ℕ
 1   { |F| ∈ ℕ }
 2  W := F;
 3   { |W| ∈ ℕ }
 4  V := ∅;
 5   { |V| ∈ ℕ ∧ |W| ∈ ℕ }
 6   { inv: Inv_fin }  { bd: (V, W) }
 7  while W ≠ ∅ do
 8      α := (V, W);
 9       { W ≠ ∅ ∧ Inv_fin ∧ (V, W) = α }
10      x := select(W);
11       { x ∈ W ∧ Inv_fin ∧ (V, W) = α }
12      W := W \ { x } ;
13       { x ∉ W ∧ Inv_fin ∧ (V, W ∪ { x } ) = α }
14      if x ∉ ↑V then
15            { x ∉ ↑V ∧ x ∉ W ∧ Inv_fin ∧ (V, W ∪ { x } ) = α }
16          V := V ∪ { x } ;
17            { x ∈ V ∧ x ∉ ↑(V \ { x } ) ∧ Inv_fin
18              ∧ (V \ { x } , W ∪ { x } ) = α }
19          W := W ∪ pb(x)          /* Note: pb(x) is finite. */
20            { x ∈ V ∧ x ∉ ↑(V \ { x } ) ∧ Inv_fin
21              ∧ (V \ { x } , (W ∪ { x } ) \ pb(x)) = α }
            /* See Lemma 3.9.                                  */
22            { Inv_fin ∧ (V, W) <^bbr_lex α }
23      else
24            { x ∈ ↑V ∧ x ∉ W ∧ Inv_fin ∧ (V, W ∪ { x } ) = α }
            /* See Lemma 3.10.                                 */
25            { Inv_fin ∧ (V, W) <^bbr_lex α }
26          skip
27            { Inv_fin ∧ (V, W) <^bbr_lex α }
28      fi
29        { Inv_fin ∧ (V, W) <^bbr_lex α }
30  od
31   { W = ∅ ∧ Inv_fin }
32   { true }
```

Figure 3.3.: Proof outline for termination of
Alg. 2.1 on p. 30.

In Fig. 3.3 we present the full proof outline for termination of the basic BR which is used in the following proposition. In the comment at the top of the figure, the abbreviation $Inv_{fin}(V,W)$ is listed.

Proposition 3.11. Given a WSTS $(S, \rightarrow, \preceq)$ with decidable $\preceq$ and effective pred-basis, the basic BR terminates for any finite set $F \subseteq S$. Formally,

$$\vdash_{TW} \{\, |F| \in \mathbb{N} \,\} \text{ Basic BR } \{\, \mathbf{true} \,\} \ .$$

Proof. By the proof outline in Fig. 3.3 and Lemma 2.12 on p. 48 (proof outlines imply correctness theorems). □

3.1.3. Total Correctness

The decomposition lemma allows to combine both partial correctness and termination to deduce total correctness of the basic backward reachability analysis.

Theorem 3.12. Consider a WSTS $(S, \rightarrow, \preceq)$ with decidable $\preceq$ and effective pred-basis. When executed with a finite set $F \in S$, the basic BR as shown in Alg. 2.1 on p. 30 terminates with finite result set V s.t. the upward-closure of V is the set of predecessors of the upward-closure of F, i.e. $\uparrow V = pre^*(\uparrow F)$. Formally,

$$\models_{tot} \{\, |F| \in \mathbb{N} \,\} \text{ Basic BR } \{\, |V| \in \mathbb{N} \wedge \uparrow V = pre^*(\uparrow F) \,\} \ .$$

Proof. The total correctness of the basic backward reachability analysis (Alg. 2.1 on p. 30) follows from Proposition 3.7 (partial correctness of the basic BR) and Proposition 3.11 (termination of the basic BR), together with Def. 2.23 on p. 46 (decomposition rule). □

For any WSTS and finite set of initial states I and any finite set of final states F, the basic BR delivers a finite result set V, s.t. $\uparrow V = pre^*(\uparrow F)$. By testing for each initial state y in I successively, whether V contains some state x with $x \preceq y$, we answer the corresponding coverability problem $I \cap pre^*(\uparrow F) \stackrel{?}{=} \emptyset$.

The basic backward reachability analysis is the foundation of our framework.

3.2. Extending the Basic BR

In this section, we work towards our algorithmic framework by gradually extending the basic backward reachability analysis. The full framework algorithm together with a description of how to instantiate it will be introduced in the following section. We will not prove the claims we make about the intermediate algorithms, but we will show total correctness of the framework algorithm later in this chapter.

Extension I: Minimization

The first extension exploits the fact that if state x is covered by state y, i.e. $x \preceq y$, first selecting x from W in a loop iteration and then y from W in a later iteration yields the same result: since x is already in V, state y obviously is in $\uparrow V$ and therefore no further update on V and W is performed. By leaving out y the algorithm's performance benefits both in terms of memory consumption and runtime—y does not have to be stored and there is at least one less loop iteration.
The idea is to simply make W a minimal basis, i.e. it only contains minimal elements w.r.t. $\preceq$. Furthermore, the same principle can be applied to V, as we are only interested in the upward-closure of V and the upward-closure of any basis of a UCS is the UCS itself, i.e. $\uparrow \text{basis}(\uparrow V) = \uparrow V$. Since set V is only accessed in the sense of a (finite) basis of a UCS—for example in the test $x \in \uparrow V$—, we can make it a *minimal* basis. Therefore, we apply the same minimization mechanism as for set W.
In Alg. 3.1, we employ a function *minimize*$(\cdot)$ that simply reduces a finite set of states to a finite set of *minimal* states w.r.t. $\preceq$.

Extension II: Disjoint $\uparrow V$ and W

During the analysis with Alg. 3.1, whenever a predecessor basis is added to W which has a non-empty intersection with $\uparrow V$ superfluous loop iterations may occur. Consider states x' and $x \in \uparrow V \cap pb(x')$. Clearly, when x is selected and removed from W no further update on V and W is executed.
The second extension, shown in Alg. 3.2, elaborates on this fact and avoids the addition of such states to W. Therefore, it subtracts $\uparrow V$ from the predecessor basis that is added to W. Due to the property that W

Input : WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$, a finite set of states $F \subseteq S$
Output : A finite set $V \subseteq S$, s.t. $\uparrow V = \mathit{pre}^*(\uparrow F)$
Comment: Given a finite base F, the algorithm computes a finite basis V of backward reachable states.

```
1  W := minimize(F);
2  V := ∅;
3  while W ≠ ∅ do
4      x := select(W);
5      W := W \ { x } ;
6      if x ∉ ↑V then
7          V := minimize(V ∪ { x } );
8          W := minimize(W ∪ pb(x))
9      fi
10 od
```

Algorithm 3.1: BR Extension I: Minimization.

contains only minimal states w.r.t. $\preceq$, the removal of some x from W leaves W and $\uparrow(V \cup \{ x \})$ disjoint. Therefore, W and $\uparrow V$ are disjoint during the whole computation and the condition $x \notin \uparrow V$ that was used in the basic BR can be omitted.

Input : WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$, a finite sets of states $F \subseteq S$
Output : A finite set $V \subseteq S$, s.t. $\uparrow V = \mathit{pre}^*(\uparrow F)$
Comment: Given a finite base F, the algorithm computes a finite basis V of backward reachable states.

```
1  W := minimize(F);
2  V := ∅;
3  while W ≠ ∅ do
4      x := select(W);
5      W := W \ { x } ;
6      V := minimize(V ∪ { x } );
7      W := minimize(W ∪ (pb(x) \ ↑V))
8  od
```

Algorithm 3.2: BR Extension II: Disjoint $\uparrow V$ and W.

In Example 2.5 on p. 28 we said that in the context of example Petri net N_{ex}, predecessors of m_Ω that cover m_Ω are superfluous for the search algorithm and can safely be ignored. This follows from the argument for this extension, as state m_Ω is added to V prior to the addition of $pb(m_\Omega) \setminus \uparrow V$ to set W. Therefore, all predecessor states that cover m_Ω are left out from being added to W and they might as well be left out in the computation of $pb(m_\Omega)$.

Extension III: Shortcut

The third extension of the basic BR is presented in Alg. 3.3. It differs from the previous algorithm by introduction of a shortcut, terminating the loop as soon as a state in the intersection of $pre^*(\uparrow F)$ and $\downarrow I$ is found. The rationale is that the existence of such a state x suffices to deduce that some initial state y with $y \succeq x$ is in $I \cap pre^*(\uparrow F)$. As this intersection is non-empty, the coverability problem is answered positively.

The shortcut works as follows. After selection and removal of a state x from W, a test $x \in \downarrow I$ is performed. If the result is positive, set V is set to only contain x and set W is emptied. With W being the empty set, the loop condition is violated and the algorithm terminates. In this case, from the result V, the set of initial states from which $\uparrow F$ can be reached can be calculated to be $\uparrow V \cap I$. If the shortcut is not taken, the algorithm works just as Alg. 3.2.

Extension IV: Explicit Inner Loop

Most of the set operations in the previous algorithms have operands consisting of an arbitrary set and a singleton, but the statement with which elements are added to W does not have a singleton operand: $W \cup (pb(x) \setminus \uparrow V)$ (earlier $W \cup pb(x)$). In Alg. 3.4 we exchange this operation by an explicit loop that checks for each state in $pb(x)$ if it does not belong to $\uparrow V$ and, in that case, adds it to W.

The operation $W := minimize(W \cup (pb(x) \setminus \uparrow V))$ is written explicitly with the use of a *foreach loop*. In the context of while programs (Def. 2.19 on p. 43) the loop

$$\textbf{foreach } x \in X \textbf{ do } S \textbf{ od}$$

Input : WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$, finite sets of states $I, F \subseteq S$
Output : A finite set $V \subseteq S$ as a solution to **Cov**.
Comment: Given finite bases I and F, if $I \hookrightarrow F$, then the algorithm computes V, s.t. $V \cap {\downarrow} I \neq \emptyset$, else the algorithm computes a finite basis V of backward reachable states.

```
1  W := minimize(F);
2  V := ∅;
3  while W ≠ ∅ do
4      x := select(W);
5      W := W \ { x } ;
6      if x ∈ ↓I then
7          V := { x } ;
8          W := ∅
9      else
10         V := minimize(V ∪ { x } );
11         W := minimize(W ∪ (pb(x) \ ↑V))
12     fi
13 od
```

Algorithm 3.3: BR Extension III: Shortcut.

is an abbreviation for

$$Y := X;\ \textbf{while}\ Y \neq \emptyset\ \textbf{do let}\ x \in Y;\ Y := Y \setminus \{\, x \,\};\ S\ \textbf{od}\,,$$

where the variable $Y \neq X$ does neither occur in S nor in the rest of the program. The *nondeterministic assignment* statement **let** $x \in Y$ means that x takes any value in Y which closely corresponds to what $\mathit{select}(Y)$ does. However, in our framework, the selection function will be allowed to guide the analysis by selecting specific states from W, whereas **let** $x \in Y$ is not used to guide the analysis.

Definition 3.2 (Nondeterministic Assignment). Given a non-empty set X, the *nondeterministic assignment* statement **let** $x \in X$ has the effect of x nondeterministically taking some value in X, i.e. $x \in X$ holds after execution of the statement. $\Diamond$

Input : WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$, finite sets of states $I, F \subseteq S$
Output : A finite set $V \subseteq S$ as a solution to **Cov**.
Comment: Given finite bases I and F, if $I \hookrightarrow F$, then the algorithm computes V, s.t. $V \cap \downarrow I \neq \emptyset$, else the algorithm computes a finite basis V of backward reachable states.

```
1  W := minimize(F);
2  V := ∅;
3  while W ≠ ∅ do
4      x := select(W);
5      W := W \ { x } ;
6      if x ∈ ↓I then
7          V := { x } ;
8          W := ∅
9      else
10         V := minimize(V ∪ { x } );
11         foreach y ∈ pb(x) do
12             if y ∉ ↑V then
13                 W := minimize(W ∪ { y } )
14             fi
15         od
16     fi
17 od
```

Algorithm 3.4: BR Extension IV: Explicit Inner Loop.

The net effect of the explicit loop for adding elements $pb(x) \setminus \uparrow V$ to W and minimizing W is the same as performing the one higher-level set operation and minimizing W. The algorithm's net behaviour is not altered by this change.

A benefit of replacing the higher-level set operation with this explicit inner loop is, however, that for an instantiation of the algorithmic framework, only a small set of simple operations on data structures has to be implemented. In fact, with the set of general data structures we provide, it involves only very little implementation effort to get efficient operations on upward-closed sets for our framework algorithm (see Ch. 6 on p. 160).

Extension V: Optimized Predecessors

For the next extension, observe that each of the previous algorithms performed a single step backward in every iteration by use of the predecessor basis *pb*. However, the result set V is either a finite basis of the predecessors of $\uparrow F$ or it contains states that are covered by initial states. Thus, the limitation to the computation of one-step predecessors during the analysis may be loosened.

Input : WSTS $\mathcal{S} = (S, \rightarrow, \preceq)$, finite sets of states $I, F \subseteq S$
Output : A finite set $V \subseteq S$ as a solution to **Cov**.
Comment: Given finite bases I and F, if $I \hookrightarrow F$, then the algorithm computes V, s.t. $V \cap \downarrow I \neq \emptyset$, else the algorithm computes a finite basis V of backward reachable states.

```
1  W := minimize(F);
2  V := ∅;
3  while W ≠ ∅ do
4      x := select(W);
5      W := W \ { x } ;
6      if x ∈ ↓I then
7          V := { x } ;
8          W := ∅
9      else
10         V := minimize(V ∪ { x } );
11         foreach y ∈ opb(x) do
12             if y ∉ ↑V then
13                 W := minimize(W ∪ { y } )
14             fi
15         od
16     fi
17 od
```

Algorithm 3.5: BR Extension V: Optimized Predecessors.

We relax the predecessor computation by use of the predecessor basis function *pb* to a function *opb* which returns a basis of *optimized* predecessors. In Sect. 3.4, we discuss the properties of *opb* in more detail, but

for now, we give a rough intuition for the function. Consider a finite set of states I and some state x, then an optimized predecessor basis is a set with

1. the optimized predecessors of x are proper predecessors of $\uparrow x$, i.e. set $opb(x)$ is a finite subset of $pre^+(\uparrow x)$,
2. if x is coverable from I via a non-empty transition sequence, then the optimized predecessors are coverable from I, i.e. $x \notin \downarrow I \wedge I \hookrightarrow x$, then $I \hookrightarrow opb(x)$, and
3. if x is not coverable from I, then the optimized predecessors are not coverable from I, i.e. $I \not\hookrightarrow x$, then $I \not\hookrightarrow opb(x)$.

The idea is that in the computation of predecessors of state x, the algorithm is allowed to take leaps over portions of the search space if the resulting answer to the coverability problem remains unaffected.
The difference between the previous extension and the one presented in Alg. 3.5 simply is the substitution of opb for pb. In Sect. 3.4 we will learn that property 3 can be omitted in the context where witness traces are available.

Extension VI: Witness Traces

This extension lifts the algorithm from answers to the **Cov** problem to the **LCov** problem, i.e. if F is coverable from I, then a transition sequence $\sigma \in L^*$ has to be presented with $I \overset{\sigma}{\hookrightarrow} F$. The ability to store a transition sequence is accomplished by the introduction of a function $T : S \longrightarrow L^*$ which is used to keep track of traces from each explored state to some final state in $\uparrow F$ and a witness function wit that delivers traces in correspondence with the optimized predecessor function, s.t. $wit_x(y)$ returns a covering trace from y to x, i.e. $y \xhookrightarrow{wit_x(y)} x$.
In Alg. 3.6 the function T is initialized to return $\perp$ for all states, which represents "no trace". From a programming perspective, function T returns $\perp$ for any state where it is not instructed to return a proper trace. The next step sets function T to return the empty trace ε for all states in W which is the minimal base of final states $\uparrow F$ at that time—of course, for any final state $x \in \uparrow F$ we expect $x \overset{\varepsilon}{\hookrightarrow} F$ to hold. We choose the

semantics of an assignment to function T to be fixed by the following definition. It is closely related to the notion of substitutions in [ABO09].

Definition 3.3 (Function Assignment). Given a variable T which represents a function $T : X \longrightarrow Y$, we denote an update to the function by

$$T(Z) := y$$

where Z is an arbitrary subset of the (potentially infinite) domain X and y is an element of target set Y. The assignment denotes a function $T' : X \longrightarrow Y$ with

$$T'(x) = \begin{cases} y & \text{if } x \in Z \\ T(X) & \text{otherwise} \end{cases} .$$

If set Z is a singleton, we omit the curly braces and write $T(z) := y$ instead of $T(\{ z \}) := y$. ♢

Remark 3.1 (Operational View on the Function Assignment). We employ an operational perspective on the notion of function assignments. In general, an assignment $T(Z) := y$ takes a number of computation steps proportional to the cardinality of the set Z. In particular, if Z is infinite, the assignment would take an infinite number of steps. There is one exception: if Z coincides with the function's domain X, the number of steps is still finite. Technically, we consider the function to have a *default* value and a distinct mapping from a finite subset Z of the domain to the codomain. Every element not in the finite subset, i.e. each element in $X \setminus Z$, is mapped to the default value. If the function is updated for the whole domain, the default value is updated and the mappings for the finite subset Z are removed. In case the function is updated on a finite set, the mappings are updated accordingly but the default value remains unchanged. ♢

After the test that $x \notin \downarrow I$, the algorithm proceeds as before until all optimized predecessors are added to the search. As soon as a new state y, a predecessor of x, is added to set W, we have to update the function T to return a trace from y to the final states s.t. $y \xhookrightarrow{T(y)} F$ holds.

Since we introduced the witness function with $y \xhookrightarrow{wit_x(y)} x$ and we already have a trace $T(x)$ from x to the final states, we can compose the two transition sequences and update function T to return $wit_x(y) \cdot T(x)$ for

state y. Notice that the trace from y to x is composed before $T(x)$ as search proceeds backwards

At the end of the computation, function T holds traces to the final states for every state in V, thus answering the **LCov** problem.

The difference between T and the wit_x function lies in their purpose and scope: When instantiating the algorithm, functions *opb* and *wit* have to be implemented in close correspondence, s.t. the witness function gives traces to some state from its optimized predecessor. Usually, the witness function comes as a byproduct of the optimized predecessor function and is computed in a—so to say—local scope. In contrast, function variable T acts as a memory where the results of the various calls to the witness function are combined to form longer traces—from the perspective of the whole algorithm, this happens in a global scope.

Extension VII: Drop Obsolete Witness Traces

With the previous algorithm, we observe that for any state that was added to W, a trace to the final states is stored and never removed. At the end, T contains traces for every state that was explored—which may be even more states than are stored in V as it is a minimal basis. While this works, it certainly uses more memory than necessary. Upon closer inspection, we only need to store traces for states that are in W and—in one special case—for the one state that is added to V if a F is found to be coverable from I.

We therefore introduce some amount of cleaning up by "dropping" values from T when they are no longer needed. Cleaning up the trace for some state x is accommodated by assignments $\tau := T(x); T(x) := \bot$ which resets the function to return $\bot$ for state x whenever state x is removed from W. We keep the trace in a temporary variable τ as we may need it for composition of traces of predecessors that are added to W. For predecessor y of x, the trace then is $wit_x(y) \cdot \tau$. And while we are at it, we may as well limit the assignment $T(y) := wit_x(y) \cdot \tau$ to those cases where there is no shorter trace currently stored for y, i.e. if $T(y) = \bot$ or $|T(y)| > |\, wit_x(y) \cdot \tau|$.

With this final extension of the procedure, we arrive at our algorithmic framework.

Input : WSLTS $\mathcal{S} = (S, L, \rightarrow, \preceq)$, finite sets of states $I, F \subseteq S$

Output : A finite set $V \subseteq S$ and a function $T : S \longrightarrow L^* \cup \{\perp\}$, a solution to **LCov**.

Comment: Given finite bases I and F, if $I \hookrightarrow F$, then the algorithm computes $\emptyset \neq V \subseteq {\downarrow} I$ and T s.t. $\forall x \in V : x \xhookrightarrow{T(x)} F$, else the algorithm computes a finite base V of backward reachable states.

1 $W := \mathit{minimize}(F)$;
2 $T(S) := \perp$;
3 $T(W) := \varepsilon$;
4 $V := \emptyset$;
5 **while** $W \neq \emptyset$ **do**
6     $x := \mathit{select}(W)$;
7     $W := W \setminus \{x\}$;
8   **if** $x \in {\downarrow} I$ **then**
9         $V := \{x\}$;
10         $W := \emptyset$
11   **else**
12         $V := \mathit{minimize}(V \cup \{x\})$;
13   **foreach** $y \in \mathit{opb}(x)$ **do**
14   **if** $y \notin {\uparrow} V$ **then**
15                 $W := \mathit{minimize}(W \cup \{y\})$;
16                 $T(y) := \mathit{wit}_x(y) \cdot T(x)$
17   **fi**
18   **od**
19   **fi**
20 **od**

Algorithm 3.6: BR Extension VI: Witness Traces.

3.3. Framework

Our algorithmic framework for backward reachability analysis is depicted in Alg. 3.7. It is parametrized in three functions.

(1) The Function *select* picks an element to explore and allows for guided search.

(2) The optimized predecessor function *opb* constructs the search tree.

(3) It comes together with a witness function *wit* that holds the sequences of labels connecting states to their optimized predecessors.

Therefore, the algorithmic framework is called $back(select, opb, wit)$. We shall refer to the pair (opb, wit) as a *search space construction* (SSC), provided it satisfies the constraints discussed in Sect. 3.4.
An SSC combined with a selection function *select* is also called an adequate instantiation of $back(select, opb, wit)$. We will show that it forms a decision procedure for coverability in WSTSs.

Definition 3.4 (Minimization Function). The minimization function reduces a basis X of a UCS to a minimal basis s.t. $minimize(X)$ is a subset of X and that $\uparrow minimize(X) = \uparrow X$ holds. ◊

Algorithm $back(select, opb, wit)$ (Alg. 3.7) uses three main variables: finite sets of states V, W, and the function T that associates states with transition sequences (or $\perp$). Technically, the T function can be implemented in several ways, for example as a hash map or by augmenting states with a corresponding trace, while finite bases V and W have to allow for certain efficient operations which we will discuss in Ch. 6 on p. 160, where we present general abstract data structures that are easy to use when instantiating the framework.
We use the three main variables in the following way. Set V is the basis of the currently explored part of the search space backward reachable from $\uparrow F$. Set W is the basis of certain predecessor states of $\uparrow V$ that have to be explored. Function T maps a state y to either $\perp$ or to a sequence σ s.t. $y \overset{\sigma}{\hookrightarrow} F$ – for states in W, function T does not map to $\perp$ but a sequence. So if there exists a path from $\downarrow I$ to $\uparrow F$, the algorithm terminates in a state where $x \in \downarrow I$ and sequence $\sigma = T(x)$ with $x \overset{\sigma}{\hookrightarrow} F$ is determined. The algorithm checks for hitting $\downarrow I$ rather than I. The

Input : WSLTS $\mathcal{S} = (S, L, \rightarrow, \preceq)$, finite sets of states $I, F \subseteq S$

Output : A finite set $V \subseteq S$ and a function $T : S \longrightarrow L^* \cup \{\perp\}$, a solution to **LCov**.

Comment: The algorithm computes set V and function T s.t. $\forall x \in V \cap {\downarrow} I : x \xhookrightarrow{T(x)} F$. The set V is non-empty if and only if $I \hookrightarrow F$.

```
1  W := minimize(F);
2  T(S) := ⊥;
3  T(W) := ε;
4  V := ∅;
5  while W ≠ ∅ do
6      x := select(W);
7      τ := T(x);
8      W := W \ { x } ;
9      T(x) := ⊥;
10     if x ∈ ↓I then
11         V := { x } ;
12         T(W) := ⊥;
13         T(x) := τ;
14         W := ∅
15     else
16         V := minimize(V ∪ { x } );
17         foreach y ∈ opb(x) do
18             if y ∉ ↑V then
19                 W := minimize(W ∪ { y } );
20                 σ := wit_x(y) · τ;
21                 if T(y) = ⊥ ∨ |T(y)| > |σ| then
22                     T(y) := σ
23                 fi
24             fi
25         od
26     fi
27 od
```

Algorithm 3.7: Algorithmic Framework *back*(*select*, *opb*, *wit*).

simulation property of $\leq$ and I being a subset of $\downarrow I$ guarantees that $\uparrow F$ is reachable from I if and only if it is reachable from $\downarrow I$.

The algorithm proceeds as follows. In each iteration of the main loop, a state x is selected and removed from the basis W of states to be explored. For that state, the sequence returned from function T is stored in τ and T is updated to map x to $\perp$ in order to reduce the memory footprint—a definition of the semantics of an assignment to function T can be found in Def. 3.3. Sequence τ leads from x to $\uparrow F$. If x is covered by an initial state z, a solution (z, τ) to the coverability has been found and the algorithm terminates. Otherwise, x is processed: $\uparrow x$ is added to the set of processed states, $\uparrow V$, and the optimized predecessors returned by $opb(x)$ are handled. Each optimized predecessor y is checked to be new in the sense that it does not lie in the processed states $\uparrow V$. This is a deviation from the basic approach to ensure that $\uparrow V$ and W are kept disjoint (the main loop satisfies invariant $\uparrow V \cap W = \emptyset$)—to reduce the memory footprint and give a quick argument for termination as the size of $\uparrow V$ increases with every iteration. Predecessor y is then added to the set of states to be processed and function T is updated to store the shortest currently known trace from y to $\uparrow F$. If the algorithm terminates with $V \cap \downarrow I = \emptyset$, the solution to the coverability problem is $\perp$ as there is no path from $\downarrow I$ to $\uparrow F$.

In the upcoming section, we take a closer look at the properties of search space constructions.

3.4. Search Space Constructions

We present the concept of *search space construction* (SSC) for the backward coverability analysis on WSTSs. The formulation is general enough to capture *invariant-based pruning* [DRV01] (Sect. 5.2 on p. 143) and *partial-order reduction* [AJKP98] (Sect. 5.3 on p. 147) as instantiations. We also propose a new search space construction, called *backward acceleration* (Sect. 5.1 on p. 131), that is based on path-learning. It again fits the new framework.

To build SSCs we need the notion a labelling function:

Definition 3.5 (Labelling Function). By $lbl_y(x)$ we denote the *labelling function*, delivering a label of a transition step from x to $\uparrow y$:

$$lbl_y(x) \in \left\{ \ell \in L \mid x \overset{\ell}{\hookrightarrow} y \right\}. \qquad \Diamond$$

Technically, a search space construction for $back(select, opb, wit)$ is a pair (opb, wit) of functions, satisfying the conditions we describe in the upcoming sections. As we will argue, already the pair (pb, lbl) is an SSC.

Adequate instantiations of SSCs lead to decision procedures for coverability. More precisely, the instantiation is a decider that additionally computes a witness path in case coverability holds. In Ch. 5 on p. 130 we take a look at instances of SSCs.

3.4.1. Optimized Predecessors and Distance Reduction

The *optimized predecessor* function opb takes as input a state x and returns a finite set of states $opb(x)$. The idea is to replace the predecessor basis construction $pb(x)$ by $opb(x)$. For the change to be correct, we require the optimized predecessor computation to be finite and *distance reducing*. Before we turn to the formal definition of opb, we introduce our distance function.

Function $dist$ maps two sets of states to the length of the shortest path from some element in the first set to some element in the second. The function returns ∞ if no such path exists.

Definition 3.6 (Distance Function). The *distance* function is the function

$$dist : \mathcal{P}(S) \times \mathcal{P}(S) \longrightarrow \mathbb{N} \cup \{ \infty \} \text{ s.t.}$$
$$(X, Y) \mapsto \min(\{ k \in \mathbb{N} \mid X \to^k Y \} \cup \{ \infty \}),$$

where $X \to^k Y$ denotes the fact that there is a transition sequence of length k from a state in X to a state in Y. $\Diamond$

Corollary 3.13 (Some Properties of the Distance Function).

1. The distance between some set X and the empty set is ∞ as there exists no transition sequence connecting the two: $dist(X, \emptyset) = dist(\emptyset, X) = \infty$.

2. For any two sets I, F, if I contains covering predecessors of F, i.e. $I \hookrightarrow F$, then and only then the distance between I and $\uparrow F$ is finite, s.t. $dist(I, \uparrow F) \neq \infty$ holds.

3. Due to the transition relation being upward-compatible with the WQO (cf. Def. 2.6 on p. 19), $I \hookrightarrow F$ is also equivalent to $dist(\downarrow I, \uparrow F) \neq \infty$.

We use this corollary in the correctness proof of our framework.

Definition 3.7 (Optimized Predecessor Function). Given a WSTS $(S, \rightarrow, \preceq)$, a finite $I \subseteq S$ and a state $x \in S$, an *optimized predecessor function opb* satisfies the following two constraints.

1. Finiteness:

$$|\, opb(x)| \in \mathbb{N} \qquad (\mathbf{Fin})$$

2. Distance Reduction:

$$0 \neq dist(\downarrow I, \uparrow x) \neq \infty \quad \Rightarrow \quad dist(\downarrow I, \uparrow opb(x)) < dist(\downarrow I, \uparrow x) \qquad (\mathbf{Dist})$$

◊

The reader should note that during the development of an SSC, the constraint **Dist** can often be checked locally, by comparing x and $opb(x)$. Also note that for sets X, Y with $X \subseteq Y$ and some set Z, $dist(Z, Y) \leq dist(Z, X)$ is implied due to $\preceq$ being a simulation relation. We formalize this property in the following corollary.

Corollary 3.14 (Inverse Montonicity of Distance). The distance does not increase if the sets under consideration only grow. Formally, for any sets X, Y, Z with $X \subseteq Y$, the relations

1. $dist(Z, Y) \leq dist(Z, X)$ and

2. $dist(Y, Z) \leq dist(X, Z)$

hold. ◊

Finally, consider the classical predecessor basis construction, i.e. when *opb* is *pb*. Already by definition of predecessors, if $\downarrow I$ is backward reachable from $\uparrow x$, then $\uparrow opb(x)$ is closer to the downward-closure of initial states $\downarrow I$ than $\uparrow x$ is.

3.4.2. Witness Traces and Search Strategies

While the major effort for instantiating the framework lies in an adequate implementation of the optimized predecessor function, the constraints of the other two procedures *wit* and *select* are potentially more simple to comply to. This holds at least from our point of view, where we assume the use of the framework's reference implementation that we present in Sect. 7.1 on p. 192.

Witness Traces. To be useful, a search space construction should also preserve witness paths that lead to a covering state. To construct such paths we need a relation between $opb(x)$ and x. It is the task of the *witness function wit* to establish this link. Given a state $x \in S$, it provides a function $wit_x : opb(x) \longrightarrow L^*$ that explains how the optimized predecessors reach x.

Definition 3.8 (Witness Function). Given a WSLTS $(S, L, \rightarrow, \preceq)$, a finite set $I \subseteq S$, a state $x \in S$ and an optimized predecessor y of x, a *witness function* wit_x computes a path $wit_x(y) \in L^*$ that leads from y to x. Formally, *wit* has to obey the following constraint:

$$\text{for all } y \in opb(x) \text{ we have } y \xhookrightarrow{wit_x(y)} x\,. \qquad (\mathbf{Wit})$$

$\Diamond$

For the trivial *opb* implementation, *pb*, the matching implementation of *wit* is *lbl*.

Definition 3.9 (Trivial Witness Function). Consider some WSLTS $(S, L, \rightarrow, \preceq)$, a finite set $I \subseteq S$, a state $x \in S$ and a basis $pb(x)$ of one-step predecessor of x.

For any one-step predecessor $y \in pb(x)$, the *trivial witness function* lbl_x returns a transition label ℓ, s.t. there is a transition from y to x with label ℓ. Formally, $lbl_x : S \longrightarrow L$ with

$$\text{for all } y \in pb(x) \text{ we have } y \xhookrightarrow{lbl_x(y)} x\,.$$

Search Strategies. The only requirement for the selection function is that it returns a state from the finite basis of states W, i.e. $select(W) \in W$. However, choosing states from W in a certain order can be very beneficial: If F is coverable from I, selecting the next state on the shortest path (backwards) from $\uparrow F$ to $\downarrow I$ in each loop iteration doubtlessly leads to a smaller number of iterations (in general) than performing a complete breadth-first search or, even worse, first selecting all those states from W that do not lie on a path to the initial states. Also in case that F is not coverable from I, there may be states which lead to a smaller search space if they are selected earlier than other states.

The loose requirement for the selection function allows for implementation of search strategies via *guided search*. We discuss varieties of search strategies tailored for specific models as well as general strategies in Sect. 5.5 on p. 155.

3.5. Arguments for Termination and Correctness

Before we give a formal proof that the algorithmic framework solves **LCov** for every adequate instantiation of *select*, *opb*, and *wit* in the next chapter, we discuss its correctness on a higher, more intuitive level.

Termination. Leaving out the case where the shortcut $x \in \downarrow I$ is taken, termination is guaranteed as UCS $\uparrow V$ is enlarged in every iteration. This converges in a finite number of steps and only finitely many iterations take place due to the stabilization lemma (Lemma 2.9 on p. 27). If the shortcut is taken, then W is explicitly set to the empty set and the loop terminates anyway.

Partial Correctness. We distinguish two cases. In case $\uparrow F$ is reachable from $\downarrow I$, there is some $x \in F$ with $dist(\downarrow I, \uparrow x) < \infty$. From termination it follows that some element $y \leq x$ will be selected from W. By requirement **Dist** and $\uparrow x \subseteq \uparrow y$, we conclude $dist(\downarrow I, \uparrow opb(y)) < dist(\downarrow I, \uparrow y) \leq dist(\downarrow I, \uparrow x)$. We iterate the argument and derive a decreasing chain of distances in $\mathbb{N}$. By well-foundedness of $(\mathbb{N}, \leq)$, we eventually arrive in $\downarrow I$. For the reverse direction, assume $\uparrow F$ is not reachable from $\downarrow I$. The reader may be suspicious since we only pose a requirement on $opb(x)$ in case $\downarrow I$ is backward reachable from x. The reason **Dist** is sufficient is in the second constraint **Wit**. It ensures optimized predecessors indeed yield a path to $\uparrow F$. So if they hit $\downarrow I$, we would be sure about reachability. Hence, in case $\uparrow F$ is not reachable from $\downarrow I$, the optimized predecessors will not hit $\downarrow I$.
In detail, when the algorithm terminates, set V and function T satisfy

$$\begin{aligned}&(dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow \uparrow V \cap \downarrow I \neq \emptyset)\\ \wedge\, &\forall x \in V \cap \downarrow I : x \xhookrightarrow{T(x)} F\,,\end{aligned}$$

expressing that V contains a state in $\downarrow I$ if and only if it is reachable from $\uparrow F$ and T then provides a witness path.
To generate a solution to a coverability problem from $x \in V \cap \downarrow I$, some $z \in I$ is chosen which covers x. The solution then is $(z, T(x))$ as $x \xhookrightarrow{T(x)} F$ implies $z \xhookrightarrow{T(x)} F$ as $\leq$ is a simulation relation.
In the later chapters, we give well-known search space exploration techniques that fall into our framework of search space constructions and we discuss our practical contributions. But first, in the next chapter, we show that our framework behaves well in the sense of total correctness.

CHAPTER FOUR

Proof of the Algorithmic Framework

We are stuck with technology when what we really want is just stuff that works.

— Douglas Adams, *The Salmon of Doubt*

Contents

4.1 Preliminaries 88
4.2 Partial Correctness 92
4.3 Termination 110
4.4 Total Correctness 124
4.5 Differences in the Partial Correctness Proofs 125

While the high-level arguments for partial correctness and termination of the framework (Sect. 3.5 on p. 85) give an intuition why it works, in this chapter, we present a formal axiomatic proof to understand its inner workings in detail.

As with the proof for the basic BR in Sect. 3.1 on p. 52, our proof consists of proof outlines in the form introduced by Owicki and Gries

which imply the truth of corresponding Hoare-style correctness formulas [OG76, ABO09]. We distinguish between a proof of partial correctness and a proof for termination which we combine at the end of this chapter by employing the decomposition rule (Def. 2.23 on p. 46) to conclude total correctness of the framework, i.e. we show that it terminates with correct results.

Before we turn to partial correctness, in the upcoming preliminary section we introduce axioms and abbreviations for groups of statements of the algorithmic framework. We call these abbreviations *subalgorithms*. These enable us to split the proof outlines into smaller chunks with lemmas that capture correctness formulas regarding the subalgorithms. At the end of the proofs for partial correctness and termination, we present proof outlines that reference the proof outlines for the corresponding subalgorithms.

4.1. Preliminaries

For our proofs of partial correctness and termination of the algorithmic framework, we employ an abbreviation of subalgorithms that enables us to talk about different parts of the algorithm independently. In Alg. 4.1, the algorithmic framework (cf. Alg. 3.7 on p. 80) is presented with placeholders for groups of statements.

The following lemma comprises the characteristics of the minimization function. It will be used throughout the proofs of partial correctness and termination of our framework.

Lemma 4.1 (Minimize). Given a set Y, the following axiom is true in the sense of partial and total correctness.
AXIOM: MINIMIZE

$$\{\, \mathbf{true} \,\}\ X := \mathit{minimize}(Y)\ \{\, X \subseteq Y \land \uparrow X = \uparrow Y \,\}$$

Proof. The definition of the minimization function (Def. 3.4 on p. 79) states that *minimize* satisfies $\mathit{minimize}(Y) \subseteq Y \land \uparrow \mathit{minimize}(Y) = \uparrow Y$ for any set Y. □

In certain situations we do not use all the information from the minimization axiom but are only interested in the special case captured by

Input : WSLTS $\mathcal{S} = (S, L, \rightarrow, \preceq)$, finite $I, F \subseteq S$
Output : A finite finite set $V \subseteq S$ and a function $T : S \longrightarrow L^* \cup \{\perp\}$, a solution to **LCov**.
Comment: The algorithm computes set V and function T s.t. $\forall x \in V \cap \downarrow I : x \xhookrightarrow{T(x)} F$. The set V is non-empty if and only if $I \hookrightarrow F$.

```
 1 W := minimize(F);            ⎫
 2 T(S) := ⊥;                   ⎬ INIT
 3 T(W) := ε;                   ⎪
 4 V := ∅;                      ⎭
 5 while W ≠ ∅ do
 6     x := select(W);          ⎫
 7     τ := T(x);               ⎬ NEXT__STATE
 8     W := W \ { x } ;         ⎪
 9     T(x) := ⊥;               ⎭
10     if x ∈ ↓I then
11         V := { x } ;         ⎫
12         T(W) := ⊥;           ⎬ FOUND__TRACE
13         T(x) := τ;           ⎪
14         W := ∅               ⎭
15     else
16         V := minimize(V ∪ { x } );                         ⎫
17         O := opb(x);                                       ⎪
18         while O ≠ ∅ do                                     ⎪
19             let y ∈ O;                                     ⎪
20             O := O \ { y } ;                               ⎪
21             if y ∉ ↑V then                             ⎫   ⎪
22                 W := minimize(W ∪ { y } );             ⎪   ⎪
23                 σ := wit_x(y) · τ;                     ⎪   ⎬ ADD__PREDECESSORS
24                 if T(y) = ⊥ ∨ |T(y)| > |σ| then    ⎫   ⎬ PROCESS__NEW__STATE
25                     T(y) := σ                      ⎬   ⎪   ⎪
26                 else                               ⎪ UPDATE__TRACE
27                     skip                           ⎪   ⎪   ⎪
28                 fi                                 ⎭   ⎪   ⎪
29             fi                                         ⎭   ⎪
30         od                                                 ⎭
31     fi
32 od
```

Algorithm 4.1: Subalgorithms of the ALGORITHMIC FRAMEWORK.

the following corollary. It holds due to the fact that $\uparrow \mathit{minimize}(Y \cup \{ y \})$ is equal to $\uparrow(Y \cup \{ y \})$ and that $Y \cup \{ y \}$ is a subset of $\uparrow(Y \cup \{ y \})$ by definition of the upward-closure operator $\uparrow$ (Def. 2.7 on p. 20).

Corollary 4.2 (Minimize with Singleton). Given a set Y, the following axiom is true in the sense of partial and total correctness.

AXIOM: MINIMIZE (ADD SINGLETON)

$$\{ \mathbf{true} \} \; X := \mathit{minimize}(Y \cup \{ y \}) \; \{ y \in \uparrow X \} \qquad \Diamond$$

In the algorithmic framework, several assignments of the form $T(W) := \sigma$ occur, where T is a function from the states of an WSLTS to finite words over transition labels (cf. Def. 3.3 on p. 76). The following lemma introduces corresponding axioms that we use in the proofs for partial correctness and termination. As noted in Remark 3.1 on p. 76, we approach function assignments from an operational perspective. We distinguish two axioms that hold in the proof system for total correctness and one more general axiom that only holds in the proof system for partial correctness.

Lemma 4.3 (Function Assignment Axioms). Consider a variable T which represents a function $T : X \longrightarrow Y$ with a possibly infinite domain X and a possibly infinite target set Y.

In the first axiom, the function is updated on a finite subset Z of its domain. It holds in both PW and TW.

AXIOM: FUNCTION ASSIGNMENT (FINITE)

$$\begin{aligned} & \{ |Z| \in \mathbb{N} \wedge Z \subseteq X \wedge y \in Y \wedge T : X \longrightarrow Y \} \\ & T(Z) := y \\ & \{ \forall z \in Z : T(z) = y \} \end{aligned}$$

The second axiom considers the case when a *default value* is assigned, s.t. it is returned for any value in the function's domain. This axiom is true in the proof systems PW and TW.

AXIOM: FUNCTION ASSIGNMENT (DOMAIN)

$$\{ y \in Y \wedge T : X \longrightarrow Y \} \; T(X) := y \; \{ \forall z \in X : T(z) = y \}$$

The third axiom will be used as a shorthand for partial correctness proofs. Here, finiteness of Z is not required and thus, termination is not guaranteed, s.t. this axiom is only true in the system PW.

AXIOM: FUNCTION ASSIGNMENT (ARBITRARY)

$$\{\, Z \subseteq X \land y \in Y \land T : X \longrightarrow Y \,\}\ T(Z) := y\ \{\forall z \in Z : T(z) = y\,\} \quad \Diamond$$

Proof. By Def. 3.3 on p. 76 (function assignment). □

The idea behind the axioms "domain" and "finite" stems from the programming perspective: It is easily possible to implement a (hash-) table which associates keys and values. This table can be updated to associate a fixed value to some finite set of keys ("finite"). For any key that has no value associated, we simply return some default value that was initially determined ("domain"). Both actions, update finite subset of domain and update default value, can be implemented to adhere the axioms above. We introduce a last axiom in the context of assignments that abbreviates the finite function assignment in the special case of a singleton set.

Corollary 4.4. Let T be a variable which represents a function $T : X \longrightarrow Y$. The following axiom is true in both the proof systems PW and TW.

AXIOM: FUNCTION ASSIGNMENT (SINGLETON)

$$\{\, z \in X \land y \in Y \land T : X \longrightarrow Y \,\}\ T(z) := y\ \{\, T(z) = y \,\} \quad \Diamond$$

Remark 4.1 (Function Assignment Axioms and Notation). For the sake of brevity in the context of this chapter we silently assume that the prerequisites $T : X \longrightarrow Y$, $Z \subseteq X$, $z \in X$, and $y \in Y$ are satisfied and omit them in the proofs. ◊

Finally, we need a correctness formula for the nondeterministic assignment.

Lemma 4.5. Given a non-empty set X, the following axiom is true in the sense of partial and total correctness.

AXIOM: NONDETERMINISTIC ASSIGNMENT

$$\{\, X \neq \emptyset \,\}\ \textbf{let}\ x \in X\ \{\, x \in X \,\}$$

Proof. The definition of the nondeterministic assignment (Def. 3.2 on p. 72) states that **let** $x \in X$ results in $x \in X$ if X is a non-empty set. □

4.2. Partial Correctness

As the code of our algorithmic framework has roughly three times the number of lines of the basic BR and it employ more variables, it is more intricate to show that its result is an answer to the posed labelled coverability problem if it is adequately instantiated. In this section, we examine the behaviour of the framework under the assumption that a labelled coverability problem is given, consisting of a WSLTS $(S, L, \rightarrow, \preceq)$ together with a finite set I of initial states and a finite set F of final states. The partial correctness proof is divided into several lemmas and mostly consists of proof outlines for the subalgorithms that we have defined in the previous section (cf. Alg. 4.1).

Abbreviations. For the sake of clarity, we employ several abbreviations, the first of which simply captures the constraint that every sequence of transition labels associated with a state represents a valid trace from that state to the upward-closure of the final states.

$$Inv_T := \forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F \tag{4.1}$$

The second abbreviation is easier understood in four stages. (1) It includes the first abbreviation, Inv_T. (2) It demands that for certain states the function T gives a sequence of transition labels, i.e. a value different from $\bot$. (3) The abbreviation is parametrized in the sense that it concerns only states in some set Z. In our proof, this set is empty unless a state is added to W and the trace function T has not been updated. (4) Except for states in Z, the abbreviation ensures that T gives traces to $\uparrow F$ for states that have to be processed, i.e. those in W, and those with which the initial states have been reached, i.e. those in $V \cap \downarrow I$.

$$Inv_{TW}(Z) := Inv_T \wedge \forall z \in W \cup (V \cap \downarrow I) : (z \in Z \vee T(z) \neq \bot) \tag{4.2}$$

While the first two shorthands are concerned with traces, the third represents the core of correctness by relating the current state of the algorithm's variables with the overall answer to the coverability problem. It says that if the final states are coverable from the initial states then and only then (1) the set $\uparrow V$ must have a nonempty intersection with $\downarrow I$ or (2) there is a path from the set $\uparrow W \cup Z$ to the initial states and the set $\uparrow W \cup Z$ must be closer to $\downarrow I$ w.r.t. *dist* than $\uparrow V$ is. The last constraint is used to ensure that the algorithm works its way towards the initial states if and only if final states are coverable from the initial states. Just as the second abbreviation, the third is parametrized in set Z. Usually, Z is the empty set unless some state x was removed from W and we want to maintain information about it.

$$\begin{aligned} Inv_{dist}(Z) := dist(\downarrow I, \uparrow F) \neq \infty \\ \Leftrightarrow (dist(\downarrow I, \uparrow V) = 0 \\ \vee \infty \neq dist(\downarrow I, \uparrow W \cup Z) < dist(\downarrow I, \uparrow V)) \end{aligned} \tag{4.3}$$

In summary, the third abbreviation is used to guarantee that the algorithm returns a correct yes-or-no answer while the first two abbreviations are used to then present a labelled transition sequence. We introduce a fourth abbreviation considering the optimized predecessors of some state later in this section.

Overview of the Proof Outline in Fig. 4.1. The proof outline Fig. 4.1 on p. 95 is central to Proposition 4.13 on p. 108 which ties together the partial correctness of our algorithmic framework. We present this outline early in this section to provide the context for the upcoming lemmas which state suitable correctness formulas for each subalgorithm.
Our goal is to show that the invariant $Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset)$ 1. is meaningful enough to imply the algorithm's postcondition if the loop exits and that it 2. holds at loop entry.

1. If loop exits, the set W is empty. In that case the invariant implies that $\uparrow V$ contains initial states if and only if the final states F are coverable from the initial states I and that for any state x in V that is covered by an initial state, the trace from that state to $\uparrow F$ can be accessed as $T(x)$. This is exactly what is needed to

answer the corresponding labelled coverability problem. To accept this implication, we write down $Inv_{dist}(\emptyset)$ in the case that W is the empty set,

$$\begin{aligned} dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow (& dist(\downarrow I, \uparrow V) = 0 \\ & \vee \infty \neq dist(\downarrow I, \emptyset) < dist(\downarrow I, \uparrow V))\,, \end{aligned}$$

and lay our focus on the fact that $dist(\downarrow I, \emptyset)$ is ∞. As the second disjunct on the right-hand side of the equivalence is false in any case, we drop it and retain

$$dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow dist(\downarrow I, \uparrow V) = 0\,.$$

Now that we have established that the invariant together with $W = \emptyset$ in fact implies the property we want to show for the algorithm, we show that it holds in the first place—at loop entry.

2. Assume that INIT satisfies the postcondition

$$V = \emptyset \wedge (\forall z \in W : T(z) \neq \bot) \wedge Inv_T \wedge W \subseteq F \wedge \uparrow W = \uparrow F\,.$$

Due to V being empty and $dist(\downarrow I, \emptyset) = \infty$ (cf. Def. 3.6 on p. 82), the invariant $Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset)$ collapses to

$$\begin{aligned} & Inv_T \wedge (\forall z \in W : T(z) \neq \bot) \\ \wedge\ & (dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow (\infty = 0 \vee \infty \neq dist(\downarrow I, \uparrow W) < \infty))\,, \end{aligned}$$

which is equivalent to

$$\begin{aligned} & Inv_T \wedge (\forall z \in W : T(z) \neq \bot) \\ \wedge\ & (dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow \infty \neq dist(\downarrow I, \uparrow W))\,. \end{aligned}$$

Since we assumed INIT's postcondition to be true, we know that $\uparrow W$ is the same as $\uparrow F$. Thus, the invariant holds at loop entry.

Partial Correctness of the Subalgorithms. We begin the formal proof of the subalgorithms by examining the one for initialisation in Lemma 4.6. Just as with the partial correctness proof for the basic BR (cf. Sect. 3.1.1

// $Inv_T := \forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F$
// $Inv_{TW}(Z) := Inv_T \wedge \forall z \in W \cup (V \cap {\downarrow} I) : (z \in Z \vee T(z) \neq \bot)$
// $Inv_{dist}(Z) := dist({\downarrow} I, {\uparrow} F) \neq \infty$
// $\Leftrightarrow (dist({\downarrow} I, {\uparrow} V) = 0 \vee \infty \neq dist({\downarrow} I, {\uparrow} W \cup Z) < dist({\downarrow} I, {\uparrow} V))$

1 $\{\, |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \,\}$
2 INIT; `/* see Lemma 4.6 */`
3 $\{\, V = \emptyset \wedge (\forall z \in W : T(z) \neq \bot) \wedge Inv_T \wedge W \subseteq F \wedge {\uparrow} W = {\uparrow} F \,\}$
4 $\{$ **inv:** $Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$
5 **while** $W \neq \emptyset$ **do**
6 $\{\, W \neq \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$
7 NEXT__STATE; `/* see Lemma 4.7 */`
8 $\left\{\, x \xhookrightarrow{\tau} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}({\uparrow} x) \,\right\}$
9 **if** $x \in {\downarrow} I$ **then**
10 $\left\{\, x \in {\downarrow} I \wedge x \xhookrightarrow{\tau} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}({\uparrow} x) \,\right\}$
11 FOUND__TRACE `/* see Lemma 4.8 */`
12 $\{\, Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$
13 **else**
14 $\left\{\, x \notin {\downarrow} I \wedge x \xhookrightarrow{\tau} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}({\uparrow} x) \,\right\}$
15 ADD__PREDECESSORS `/* see Lemma 4.10 */`
16 $\{\, Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$
17 **fi**
18 $\{\, Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$
19 **od**
20 $\{\, W = \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$
21 $\{\, Inv_{TW}(\emptyset)$
22 $\wedge\, dist({\downarrow} I, {\uparrow} F) \neq \infty \Leftrightarrow (dist({\downarrow} I, {\uparrow} V) = 0 \vee \infty \neq dist({\downarrow} I, \emptyset)) \,\}$
23 $\{\, Inv_{TW}(\emptyset) \wedge dist({\downarrow} I, {\uparrow} F) \neq \infty \Leftrightarrow dist({\downarrow} I, {\uparrow} V) = 0 \,\}$
24 $\{\, Inv_T \wedge (\forall z \in V \cap {\downarrow} I : T(z) \neq \bot)$
25 $\wedge\, dist({\downarrow} I, {\uparrow} F) \neq \infty \Leftrightarrow dist({\downarrow} I, {\uparrow} V) = 0 \,\}$
26 $\left\{\, (\forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F) \wedge (\forall z \in V \cap {\downarrow} I : T(z) \neq \bot) \right.$
27 $\wedge\, dist({\downarrow} I, {\uparrow} F) \neq \infty \Leftrightarrow dist({\downarrow} I, {\uparrow} V) = 0 \,\}$
28 $\left\{\, (dist({\downarrow} I, {\uparrow} F) \neq \infty \Leftrightarrow {\uparrow} V \cap {\downarrow} I \neq \emptyset) \wedge \forall x \in V \cap {\downarrow} I : x \xhookrightarrow{T(x)} F \,\right\}$

Figure 4.1.: Proof outline for partial correctness of Alg. 3.7 on p. 80.

on p. 53), the precondition that ensures the input sets to be finite is dismissed as it is only actively needed to show termination. Lemma 4.1 is employed for the effect of statement $W := \mathit{minimize}(F)$ and the following assignments to function T are resolved via Lemma 4.3 (cases "domain" and "finite"). It is noteworthy that the assertion right after the assignment $T(S) := \bot$, i.e.

$$(\forall z \in S : T(z) = \bot) \wedge W \subseteq F \wedge \uparrow W = \uparrow F\,,$$

implies $\forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F$ (which we abbreviate by Inv_T) as there is no state z in S with $T(z)$ different from $\bot$. Then, after $T(W)$ is set to the empty word ε, condition Inv_T still holds due to W being a subset of the final states. Each state z in W is automatically in $\uparrow F$, s.t. $z \overset{\varepsilon}{\hookrightarrow} F$ holds. Since Inv_T is a strong enough requirement, we dispose of the information on the concrete value for $T(z)$ and simply state $\forall z \in W : T(z) \neq \bot$ in the following assertion. The effect of the last statement, $V := \emptyset$, results in meeting the postcondition for INIT that we aimed for.

Lemma 4.6 (init and Partial Correctness). The correctness formula

$$\begin{array}{l} \{\, |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \,\} \\ \text{INIT} \\ \{\, V = \emptyset \wedge (\forall x \in W : T(x) \neq \bot) \wedge \mathit{Inv}_T \wedge W \subseteq F \wedge \uparrow W = \uparrow F \,\} \end{array}$$

is true in the sense of partial correctness.

Proof. By the proof outline in Fig. 4.2, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

In the first part of the loop body, NEXT_STATE, some state x is selected from the set W of states to process and the sequence $T(x)$ is stored in a temporary variable τ. Then, the state x is removed from W and function T is updated to return the default value $\bot$ for state x. The following lemma shows that $x \overset{\tau}{\hookrightarrow} F$ holds after the assignment $\tau = T(x)$ which follows directly from $\mathit{Inv}_{TW}(\emptyset)$: in essence, it states that for any z in W, the value of $T(z)$ is a transition label sequence that leads from z to the

$$// \;\; Inv_T := \forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F$$

```
1   { |I| ∈ ℕ ∧ |F| ∈ ℕ }
2   { true }
3  W := minimize(F);
4   { W ⊆ F ∧ ↑W = ↑F }
5  T(S) := ⊥;
6   { (∀z ∈ S : T(z) = ⊥) ∧ W ⊆ F ∧ ↑W = ↑F }
7   { Inv_T ∧ W ⊆ F ∧ ↑W = ↑F }
8  T(W) := ε;
9   { (∀z ∈ W : T(z) = ε) ∧ Inv_T ∧ W ⊆ F ∧ ↑W = ↑F }
10  { (∀z ∈ W : T(z) ≠ ⊥) ∧ Inv_T ∧ W ⊆ F ∧ ↑W = ↑F }
11 V := ∅;
12  { V = ∅ ∧ (∀z ∈ W : T(z) ≠ ⊥) ∧ Inv_T ∧ W ⊆ F ∧ ↑W = ↑F }
```

Figure 4.2.: Proof outline for partial correctness – INIT.

upward-closure of F. More importantly, $Inv_{dist}(\uparrow x)$ holds after removal of x from W as $\uparrow(W \cup \{ x \})$ is equal to $\uparrow W \cup \uparrow x$.

Lemma 4.7 (next_state and Partial Correctness). The correctness formula

$$\{ W \neq \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \}$$
$$\text{NEXT_STATE}$$
$$\left\{ x \xhookrightarrow{\tau} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \right\}$$

is true in the sense of partial correctness.

Proof. By the proof outline in Fig. 4.3, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

In case a trace from the upward-closure of the final states is found backwards to the initial states, FOUND_TRACE is executed. Lemma 4.8 captures the correctness of this subalgorithm in the sense that the main loop's invariant is also the postcondition of FOUND_TRACE. The main goal of the shortcut that is taken when a state x is reached backwards

```
    // Inv_T := ∀z ∈ S : T(z) ≠ ⊥ ⇒ z --T(z)--> F
    // Inv_TW(Z) := Inv_T ∧ ∀z ∈ W ∪ (V ∩ ↓I) : (z ∈ Z ∨ T(z) ≠ ⊥)
    // Inv_dist(Z) := dist(↓I, ↑F) ≠ ∞
    // ⇔ (dist(↓I, ↑V) = 0 ∨ ∞ ≠ dist(↓I, ↑W ∪ Z) < dist(↓I, ↑V))
 1   { W ≠ ∅ ∧ Inv_TW(∅) ∧ Inv_dist(∅) }
 2 x := select(W);
 3   { x ∈ W ∧ W ≠ ∅ ∧ Inv_TW(∅) ∧ Inv_dist(∅) }
 4 τ := T(x);
 5   { τ = T(x) ∧ x ∈ W ∧ W ≠ ∅ ∧ Inv_TW(∅) ∧ Inv_dist(∅) }
 6   { x --τ--> F ∧ Inv_TW(∅) ∧ Inv_dist(∅) }
 7 W := W \ { x };
 8   { x --τ--> F ∧ Inv_TW(∅) ∧ Inv_dist(↑x) }
 9 T(x) := ⊥;
10   { T(x) = ⊥ ∧ x --τ--> F ∧ Inv_TW(∅) ∧ Inv_dist(↑x) }
11   { x --τ--> F ∧ Inv_TW(∅) ∧ Inv_dist(↑x) }
```

Figure 4.3.: Proof outline for partial correctness – NEXT__STATE.

that lies in the downward-closure of the initial states is to gracefully stop the algorithms loop. Therefore, all traces that are stored in T but the one for x are dismissed by setting $T(W) := \bot$ and $T(x) := \tau$, where τ is a sequence of transition labels from x to $\uparrow F$, and setting V to the singleton set $\{\, x \,\}$. Finally, W is cleared which leads to the main loop's condition being evaluated to **false**. If we keep in mind that x is covered by an initial state and that it is also set to be the sole element of V, the proof outline in Fig. 4.4 can be followed up to the assignment $W := \emptyset$ rather easily. There, we end up with the assertion

$$W = \emptyset \wedge {\downarrow} I \cap V = \{\, x \,\} \wedge x \xhookrightarrow{T(x)} F \wedge \mathit{Inv}_T \wedge \mathit{Inv}_{\mathit{dist}}({\uparrow} x)$$

which implies that for any state z in the intersection of V and $\downarrow I$, function $T(z)$ yields a value different from $\bot$. To be exact, there is only one

state in the intersection, x, and $T(x)$ yields τ, the trace from x to $\uparrow F$. Therefore, we can introduce the abbreviation $Inv_{TW}(\emptyset)$ and deduce the truth of the next assertion, i.e.

$$\downarrow I \cap V = \{\, x \,\} \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x)\,.$$

As we know that the intersection of V and $\downarrow I$ is nonempty, it follows that the distance between the two is zero, i.e. $dist(\downarrow I, \uparrow V) = 0$. This enables us to bring down $Inv_{dist}(\uparrow x)$ to $Inv_{dist}(\emptyset)$ and conclude that the subalgorithm FOUND_TRACE satisfies the required postcondition when its precondition is met.

Lemma 4.8 (found_trace and Partial Correctness). The correctness formula

$$\left\{\, x \in \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \,\right\}$$

FOUND_TRACE

$$\{\, Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \,\}$$

is true in the sense of partial correctness.

Proof. By the proof outline in Fig. 4.4, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

For the partial correctness proofs of the remaining subalgorithms, we introduce an abbreviation that maintains information on the set O which contains only optimized predecessors of x and a relationship between the set of optimized predecessors of x, $opb(x)$, and other variables of the algorithm.

$$\begin{aligned} Inv_{opb}(Z) = x \in \uparrow V \wedge x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge O \subseteq opb(x) \\ \wedge\ opb(x) \subseteq \uparrow W \cup \uparrow V \cup O \cup Z \end{aligned} \tag{4.4}$$

The ADD_PREDECESSORS subalgorithm iterates over the set of optimized predecessors of x. It does so by storing the optimized predecessors in a temporary variable O and successively removing elements from it—and processing those elements—until it is empty. For the inner loop,

```
// Inv_T := ∀z ∈ S : T(z) ≠ ⊥ ⇒ z ↪^{T(z)} F
// Inv_TW(Z) := Inv_T ∧ ∀z ∈ W ∪ (V ∩ ↓I) : (z ∈ Z ∨ T(z) ≠ ⊥)
// Inv_dist(Z) := dist(↓I, ↑F) ≠ ∞
// ⇔ (dist(↓I, ↑V) = 0 ∨ ∞ ≠ dist(↓I, ↑W ∪ Z) < dist(↓I, ↑V))
```

$$
\begin{array}{rl}
1 & \left\{ x \in \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \right\} \\
2 & \left\{ x \in \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
3 & V := \{ x \}; \\
4 & \left\{ V = \{ x \} \wedge x \in \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
5 & \left\{ \downarrow I \cap V = \{ x \} \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
6 & T(W) := \bot; \\
7 & \left\{ (\forall z \in W : T(z) = \bot) \wedge \downarrow I \cap V = \{ x \} \wedge x \overset{\tau}{\hookrightarrow} F \right. \\
8 & \quad \left. \wedge\ Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
9 & \left\{ \downarrow I \cap V = \{ x \} \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
10 & T(x) := \tau; \\
11 & \left\{ T(x) = \tau \wedge \downarrow I \cap V = \{ x \} \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
12 & \left\{ \downarrow I \cap V = \{ x \} \wedge x \xhookrightarrow{T(x)} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
13 & W := \emptyset \\
14 & \left\{ W = \emptyset \wedge \downarrow I \cap V = \{ x \} \wedge x \xhookrightarrow{T(x)} F \wedge Inv_{T} \wedge Inv_{dist}(\uparrow x) \right\} \\
15 & \{ \downarrow I \cap V = \{ x \} \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \} \\
16 & \{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \}
\end{array}
$$

Figure 4.4.: Proof outline for partial correctness – FOUND__TRACE.

$Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset)$ is an invariant. As mentioned, in each iteration the algorithm picks some state y from the set of optimized predecessors and runs PROCESS_NEW_STATE, after which the invariant holds again. At this level, there are two implications we need to discuss: (1) the invariant holds at loop entry and (2) the negated loop condition and the invariant together imply the postcondition $Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset)$. To accept that the invariant is satisfied at the beginning of the loop, we observe that $O = opb(x)$ implies $O \subseteq opb(x) \wedge opb(x) \subseteq \uparrow W \cup \uparrow V \cup O \cup Z$. In order to see that the postcondition is implied by $O = \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset)$, we turn to the following lemma.

Lemma 4.9. The following implication holds.

$$O = \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \quad \Rightarrow \quad Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset)$$

Proof. We substitute the abbreviations and use $O = \emptyset$ to observe that we are actually to show

$$\begin{aligned}
&\quad Inv_{TW}(\emptyset) \\
&\wedge\ (dist(\downarrow I, \uparrow F) \neq \infty \\
&\qquad \Leftrightarrow (dist(\downarrow I, \uparrow V) = 0 \vee \infty \neq dist(\downarrow I, \underline{\uparrow W \cup \uparrow x}) < dist(\downarrow I, \uparrow V)) \\
&\wedge\ x \in \uparrow V \wedge x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge opb(x) \subseteq \uparrow W \cup \uparrow V \\
\Rightarrow & \\
&\quad Inv_{TW}(\emptyset) \\
&\wedge\ (dist(\downarrow I, \uparrow F) \neq \infty \\
&\qquad \Leftrightarrow (dist(\downarrow I, \uparrow V) = 0 \vee \infty \neq dist(\downarrow I, \underline{\uparrow W}) < dist(\downarrow I, \uparrow V))\,,
\end{aligned}$$

where we have underlined the crucial detail. We prove this using a case distinction.

- Case $dist(\downarrow I, \uparrow W \cup \uparrow x) = \infty$. Here, the implication is trivially satisfied due to the inverse monotonicity of the distance function (Corollary 3.14 on p. 83) and thus $dist(\downarrow I, \uparrow W) = \infty$.

- Case $dist(\downarrow I, \uparrow W \cup \uparrow x) \neq \infty$. We show the implication by contradiction.

Assume the right-hand side of the implication is violated, i.e.

$$dist(\downarrow I, \uparrow W) \geq dist(\downarrow I, \uparrow V)$$

holds.

First, we use the inequalities

$$\underbrace{dist(\downarrow I, \uparrow W \cup \uparrow x) <}_{\text{implication's left-hand side}} dist(\downarrow I, \uparrow V) \underbrace{\leq dist(\downarrow I, \uparrow W)}_{\text{assumption}}$$

to conclude that $dist(\downarrow I, \uparrow W \cup \uparrow x) < dist(\downarrow I, \uparrow W)$ holds.

This, together with the definition of the distance function (Def. 3.6 on p. 82) and with Corollary 3.14 on p. 83 (inverse monotonicity of distance), means that the shortest path from $\downarrow I$ to $\uparrow W \cup \uparrow x$ must stem from $\uparrow x$, i.e.

$$dist(\downarrow I, \uparrow x) = dist(\downarrow I, \uparrow W \cup \uparrow x)\,.$$

From this equality and the condition $\infty \neq dist(\downarrow I, \uparrow W \cup \uparrow x)$ of the current case under consideration, it follows that x is coverable from $\downarrow I$, i.e. $\infty \neq dist(\downarrow I, \uparrow x)$.

Second, the **Dist** property (cf. Def. 3.7 on p. 83) and x not being element of $\downarrow I$, but x being coverable from $\downarrow I$, tells us that the initial states $\downarrow I$ are closer to $\uparrow opb(x)$ than they are to $\uparrow x$. Formally, $dist(\downarrow I, \uparrow opb(x)) < dist(\downarrow I, \uparrow x)$.

We now show that $opb(\uparrow x)$ is not a subset of $\uparrow W \cup \uparrow V$ which contradicts the conjunct $opb(\uparrow x) \subseteq \uparrow W \cup \uparrow V$ on the implication's left-hand side.

Let y be a state in $opb(x)$, s.t. $dist(\downarrow I, \uparrow y) < dist(\downarrow I, \uparrow x)$. The existence of such a state is guaranteed by the fact that the UCS of optimized predecessors is closer to the initial states, i.e.

$$dist(\downarrow I, \uparrow opb(x)) < dist(\downarrow I, \uparrow x)\,,$$

as there is some $y' \in \uparrow opb(x)$ that is closer than $\uparrow x$ and this y' is in the upward-closure of y. From the implication's left-hand side we know that y is in $\uparrow W$ or in $\uparrow V$.

- Assume the state y is in $\uparrow V$. By the inverse monotonicity of the distance function (Corollary 3.14 on p. 83) this implies that $\uparrow V$ at least as close to the initial states as $\uparrow y$ is, i.e.

  $$dist(\downarrow I, \uparrow V) \leq dist(\downarrow I, \uparrow y) < dist(\downarrow I, \uparrow x)\,,$$

  which implies that $\uparrow V$ is closer than $\uparrow W \cup \uparrow x$. But this contradicts the left-hand side of the implication that states

  $$dist(\downarrow I, \uparrow W \cup \uparrow x) < dist(\downarrow I, \uparrow V)\,.$$

  Hence, y is not an element of $\uparrow V$.

- Assume the state y is in $\uparrow W$. By the inverse monotonicity of the distance function (Corollary 3.14 on p. 83) this implies that $\uparrow W$ at least as close to the initial states as $\uparrow y$ is, i.e.

  $$dist(\downarrow I, \uparrow W) \leq dist(\downarrow I, \uparrow y) < dist(\downarrow I, \uparrow x)\,,$$

  which directly contradicts the fact

  $$dist(\downarrow I, \uparrow x) < dist(\downarrow I, \uparrow W)$$

  that we have established earlier in the case that the distance $dist(\downarrow I, \uparrow W \cup \uparrow x)$ is different from ∞. Hence, y is not an element of $\uparrow W$.

Therefore, y is neither part of $\uparrow W$ nor of $\uparrow V$, leaving us to conclude that $opb(x)$ is no subset of $\uparrow W \cup \uparrow V$. This is a contradiction to the left-hand side of the implication. Hence, the assumption is wrong and $\infty \neq dist(\downarrow I, \uparrow W) < dist(\downarrow I, \uparrow V)$ holds.

This concludes the proof of the implication. □

With Lemma 4.9, we are able to capture partial correctness of the sub-algorithm ADD_PREDECESSORS in the upcoming lemma.

Lemma 4.10 (add_predecessors and Partial Correctness). The correctness formula

$$\Big\{\, x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge \mathit{Inv}_{TW}(\emptyset) \wedge \mathit{Inv}_{dist}(\uparrow x) \,\Big\}$$
$$\text{ADD_PREDECESSORS}$$
$$\{\, \mathit{Inv}_{TW}(\emptyset) \wedge \mathit{Inv}_{dist}(\emptyset) \,\}$$

is true in the sense of partial correctness.

Proof. By the proof outline in Fig. 4.5, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

Lemma 4.11 captures the effect of PROCESS_NEW_STATE when it is executed in the context of ADD_PREDECESSORS, i.e. the algorithm is in a state where

$$y \in \mathit{opb}(x) \wedge \mathit{Inv}_{TW}(\emptyset) \wedge \mathit{Inv}_{dist}(\uparrow x) \wedge \mathit{Inv}_{opb}(\,\{\, y \,\}\,)$$

holds and the subalgorithm has to satisfy $\mathit{Inv}_{TW}(\emptyset) \wedge \mathit{Inv}_{dist}(\uparrow x) \wedge \mathit{Inv}_{opb}(\emptyset)$ as its postcondition if it terminates. It does so by ensuring that x's optimized predecessor y is either already contained in $\uparrow V$ or it is made sure that it is in the upward-closure of the set W of states to be processed.

First, consider the case when y is in $\uparrow V$. The conjunction of the precondition and $y \in \uparrow V$ has to directly imply the postcondition which is rather simple as the following implication, which highlights the central part of the implied postcondition, is easily accepted:

$$y \in \uparrow V \wedge \mathit{opb}(x) \subseteq \uparrow W \cup \uparrow V \cup O \cup \{\, y \,\} \quad \Rightarrow \quad \mathit{opb}(x) \subseteq \uparrow W \cup \uparrow V \cup O$$

In the case that y it not an element of $\uparrow V$, the effect of $W := \mathit{minimize}(W \cup \{\, y \,\}\,)$ is explained by Corollary 4.2 (minimize with singleton) and the **Wit** (cf. Def. 3.8 on p. 84) constraint is used to resolve the assignment $\sigma := \mathit{wit}_x(y) \cdot \tau$. The main proof obligation of this case is relayed to the last subalgorithm UPDATE_TRACE.

// $Inv_T := \forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F$
// $Inv_{TW}(Z) := Inv_T \wedge \forall z \in W \cup (V \cap \downarrow I) : (z \in Z \vee T(z) \neq \bot)$
// $Inv_{dist}(Z) := dist(\downarrow I, \uparrow F) \neq \infty$
// $\Leftrightarrow (dist(\downarrow I, \uparrow V) = 0 \vee \infty \neq dist(\downarrow I, \uparrow W \cup Z) < dist(\downarrow I, \uparrow V))$
// $Inv_{opb}(Z) := x \in \uparrow V \wedge x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge O \subseteq opb(x)$
// $\wedge\ opb(x) \subseteq \uparrow W \cup \uparrow V \cup O \cup Z$

1 $\left\{ x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \right\}$
2 $V := minimize(V \cup \{ x \});$
3 $\left\{ x \in \uparrow V \wedge x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \right\}$
4 $O := opb(x);$
5 $\{ O = opb(x) \wedge x \in \uparrow V \wedge x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F$
6 $\wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \}$
7 { **inv:** $Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset)$ }
8 **while** $O \neq \emptyset$ **do**
9 $\{ O \neq \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \}$
10 **let** $y \in O$;
11 $\{ y \in O \wedge O \neq \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \}$
12 $\{ y \in opb(x) \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \}$
13 $O := O \setminus \{ y \};$
14 $\{ y \notin O \wedge y \in opb(x) \wedge Inv_{TW}(\emptyset)$
15 $\wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\{ y \}) \}$
16 $\{ y \in opb(x) \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\{ y \}) \}$
17 PROCESS__NEW__STATE `/* see Lemma 4.11 */`
18 $\{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \}$
19 **od**
20 $\{ O = \emptyset \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \}$
`/* See Lemma 4.9. */`
21 $\{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\emptyset) \}$

Figure 4.5.: Proof outline for partial correctness – ADD__PREDECESSORS.

Lemma 4.11 (process_new_state and Partial Correctness). The correctness formula

$$\{\, y \in opb(x) \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\{\, y \,\}) \,\}$$
$$\text{PROCESS_NEW_STATE}$$
$$\{\, Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \,\}$$

is true in the sense of partial correctness.

Proof. By the proof outline in Fig. 4.6, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

In order to accept the partial correctness of the algorithmic framework, the correctness of the subalgorithm UPDATE_TRACE remains to be shown. There, the function T that associates states in W with traces to the upward-closure of final states is updated with a new trace for the optimized predecessors y under consideration if T does not already associate it with a trace or the associated trace is longer than the new trace.

We begin with the case when no update to T is performed as it already associates a shorter trace with y. For this case, we highlight the central points of the implication (between lines 13 and 16 in Fig. 4.7), where we need to show:

$$\begin{aligned} &T(y) \neq \bot \wedge (\forall z \in W \cup (V \cap {\downarrow} I) : (z \in \{\, y \,\} \vee T(z) \neq \bot)) \\ \Rightarrow\ &\forall z \in W \cup (V \cap {\downarrow} I) : (z \in \emptyset \vee T(z) \neq \bot) \end{aligned}$$

We accept this implication due to the fact that the disjunction on the left-hand side is true for $z = y$ because of $T(y) \neq \bot$ and the condition $z \in \{\, y \,\}$ can be dropped.

In case the function T is updated to associate trace σ with y, we employ Corollary 4.4 for the effect of function assignment (singleton) $T(y) := \sigma$. As $y \overset{\sigma}{\hookrightarrow} F$ holds, we are sure that Inv_T is still valid after the assignment and apply the reasoning from the above case to conclude the implication between lines 8 and 11 in Fig. 4.7. Therefore, $Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset)$ holds after UPDATE_TRACE has been executed.

$$
\begin{array}{rl}
 & \text{// } Inv_T := \forall z \in S : T(z) \neq \bot \Rightarrow z \xhookrightarrow{T(z)} F \\
 & \text{// } Inv_{TW}(Z) := Inv_T \wedge \forall z \in W \cup (V \cap \downarrow I) : (z \in Z \vee T(z) \neq \bot) \\
 & \text{// } Inv_{dist}(Z) := dist(\downarrow I, \uparrow F) \neq \infty \\
 & \text{// } \Leftrightarrow (dist(\downarrow I, \uparrow V) = 0 \vee \infty \neq dist(\downarrow I, \uparrow W \cup Z) < dist(\downarrow I, \uparrow V)) \\
 & \text{// } Inv_{opb}(Z) := x \in \uparrow V \wedge x \notin \downarrow I \wedge x \overset{\tau}{\hookrightarrow} F \wedge O \subseteq opb(x) \\
 & \text{// } \qquad\qquad \wedge\ opb(x) \subseteq \uparrow W \cup \uparrow V \cup O \cup Z \\
1 & \{ y \in opb(x) \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\{ y \}) \} \\
2 & \textbf{if } y \notin \uparrow V \textbf{ then} \\
3 & \quad \{ y \notin \uparrow V \wedge y \in opb(x) \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \\
4 & \quad\quad \wedge Inv_{opb}(\{ y \}) \} \\
5 & \quad W := minimize(W \cup \{ y \}); \\
6 & \quad \{ y \in \uparrow W \wedge y \notin \uparrow V \wedge y \in opb(x) \\
7 & \quad\quad \wedge Inv_{TW}(\{ y \}) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\
8 & \quad \sigma := wit_x(y) \cdot \tau; \\
9 & \quad \{ \sigma = wit_x(y) \cdot \tau \wedge y \in \uparrow W \wedge y \notin \uparrow V \wedge y \in opb(x) \\
10 & \quad\quad \wedge Inv_{TW}(\{ y \}) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\
11 & \quad \Big\{ y \overset{\sigma}{\hookrightarrow} F \wedge y \in \uparrow W \wedge y \notin \uparrow V \wedge y \in opb(x) \\
12 & \quad\quad \wedge Inv_{TW}(\{ y \}) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\
13 & \quad \text{UPDATE_TRACE} \qquad\qquad \texttt{/* see Lemma 4.12 */} \\
14 & \quad \{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\
15 & \textbf{else} \\
16 & \quad \{ y \in \uparrow V \wedge y \in opb(x) \wedge Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \\
17 & \quad\quad \wedge Inv_{opb}(\{ y \}) \} \\
18 & \quad \{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\
19 & \quad \textbf{skip} \\
20 & \quad \{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\
21 & \textbf{fi} \\
22 & \{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \}
\end{array}
$$

Figure 4.6.: Proof outline for partial correctness – PROCESS_NEW_STATE.

Lemma 4.12 (update_trace and Partial Correctness). The correctness formula

$$\begin{aligned} &\Big\{ y \overset{\sigma}{\hookrightarrow} F \wedge y \in \uparrow W \wedge y \notin \uparrow V \wedge y \in opb(x) \\ &\quad \wedge Inv_{TW}(\{ y \}) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \\ &\textsc{update_trace} \\ &\{ Inv_{TW}(\emptyset) \wedge Inv_{dist}(\uparrow x) \wedge Inv_{opb}(\emptyset) \} \end{aligned}$$

is true in the sense of partial correctness.

Proof. By the proof outline in Fig. 4.7, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

Without further ado, we are able to present the following proposition which expresses the partial correctness of our algorithmic framework.

Proposition 4.13. Consider $(S, L, \rightarrow, \preceq)$ a WSLTS with a decidable $\preceq$ and an effective pred-basis. Let *select*, *opb*, *wit* be adequately instantiated. If the algorithmic framework is run with finite sets $I, F \subseteq S$ as input, it terminates with a finite result V, T, s.t. if and only if F is coverable from I, i.e. $I \hookrightarrow F$, then $\uparrow V \cap \downarrow I$ is non-empty. For each state x in the intersection of V and $\downarrow I$, $T(x)$ is a trace from x to $\uparrow F$. Formally,

$$\begin{aligned} \vdash_{PW} \quad &\{ |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \} \\ &\textsc{Algorithmic Framework} \\ &\big\{ (dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow \uparrow V \cap \downarrow I \neq \emptyset) \\ &\quad \wedge \forall x \in V \cap \downarrow I : x \xhookrightarrow{T(x)} F \big\} . \end{aligned}$$

Proof. By Corollary 3.13 on p. 82, the finiteness of $dist(\downarrow I, \uparrow F)$ is equivalent to F being coverable from I. By the proof outline in Fig. 4.1 on p. 95 together with Lemma 4.6 (init), Lemma 4.7 (next_state), Lemma 4.8 (found_trace), Lemma 4.10 (add_predecessors), Lemma 4.11 (process_new_state), Lemma 4.12 (update_trace) and Lemma 2.12 on p. 48 (proof outlines imply correctness theorems). □

```
// Inv_T := ∀z ∈ S : T(z) ≠ ⊥ ⇒ z ↪^{T(z)} F
// Inv_TW(Z) := Inv_T ∧ ∀z ∈ W ∪ (V ∩ ↓I) : (z ∈ Z ∨ T(z) ≠ ⊥)
// Inv_dist(Z) := dist(↓I, ↑F) ≠ ∞
// ⇔ (dist(↓I, ↑V) = 0 ∨ ∞ ≠ dist(↓I, ↑W ∪ Z) < dist(↓I, ↑V))
// Inv_opb(Z) := x ∈ ↑V ∧ x ∉ ↓I ∧ x ↪^τ F ∧ O ⊆ opb(x)
//                ∧ opb(x) ⊆ ↑W ∪ ↑V ∪ O ∪ Z
```

1 $\Big\{ y \overset{\sigma}{\hookrightarrow} F \land y \in \uparrow W \land y \notin \uparrow V \land y \in opb(x)$

2 $\quad \land\, Inv_{TW}(\{y\}) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

3 **if** $T(y) = \bot \lor |T(y)| > |\sigma|$ **then**

4 $\quad \{ (T(y) = \bot \lor |T(y)| > |\sigma|)$

5 $\quad\quad \land\, y \overset{\sigma}{\hookrightarrow} F \land y \in \uparrow W \land y \notin \uparrow V \land y \in opb(x)$

6 $\quad\quad \land\, Inv_{TW}(\{y\}) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

7 $\quad T(y) := \sigma$

8 $\quad \{ T(y) = \sigma$

9 $\quad\quad \land\, y \overset{\sigma}{\hookrightarrow} F \land y \in \uparrow W \land y \notin \uparrow V \land y \in opb(x)$

10 $\quad\quad \land\, Inv_{TW}(\{y\}) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

11 $\quad \{ Inv_{TW}(\emptyset) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

12 **else**

13 $\quad \{ \neg(T(y) = \bot \lor |T(y)| > |\sigma|)$

14 $\quad\quad \land\, y \overset{\sigma}{\hookrightarrow} F \land y \in \uparrow W \land y \notin \uparrow V \land y \in opb(x)$

15 $\quad\quad \land\, Inv_{TW}(\{y\}) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

16 $\quad \{ Inv_{TW}(\emptyset) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

17 $\quad$ **skip**

18 $\quad \{ Inv_{TW}(\emptyset) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

19 **fi**

20 $\{ Inv_{TW}(\emptyset) \land Inv_{dist}(\uparrow x) \land Inv_{opb}(\emptyset) \}$

Figure 4.7.: Proof outline for partial correctness – UPDATE__TRACE.

Now that we established that the framework solves the labelled coverability problem if the algorithm returns a result, we face the task to prove that it terminates for any valid input.

4.3. Termination

In order to show termination of the algorithmic framework, we have to employ a term that takes values in a set that is well-founded w.r.t. some ordering. This termination term has to decrease in value with each loop iteration. The lexicographical ordering $>_{lex}^{bbr}$ (Def. 3.1 on p. 63) that we used to show termination of the basic BR does not suffice as the UCS $\uparrow V$ does not grow monotonically with each iteration: If the shortcut $x \in \downarrow I$ is taken when the initial states are reached, V is set to $\{ x \}$ which is disjoint to the previous value of V. However, up to the iteration when the shortcut is taken, the ordering for termination of the basic BR is fine. Remember that when the shortcut is taken, the loop terminates immediately, as W is emptied and up to that point in time, sets I and $\uparrow V$ are disjoint. Therefore, our idea for a well-founded ordering to show termination of the framework augments the ordering $>_{lex}^{bbr}$ (Def. 3.1 on p. 63) by a proper subset relation on $I \setminus \uparrow V$ that ensures the strict decrease of the termination term in case the shortcut is taken: Unless the shortcut is taken, $I \setminus \uparrow V$ is I as the sets are disjoint. If the shortcut is taken, then there is an $x \in \downarrow I$—which means that there is an $x' \in I$ with $x \preceq x'$—and $I \setminus \uparrow x$ is a proper subset of I.

Definition 4.1 (Lexicographical Order for Termination of the Framework). Given a QO $\preceq$, we define the lexicographical order $>_{lex}^{term}$ on triplet of finite sets s.t. the upward-closures of the first components are in proper subset relation or they are equal and the pairs consisting of the second and third components are in relation $>_{lex}^{bbr}$. Formally, for tuples of finite sets (I, V, W) and (I', V', W'), we define

$$
\begin{aligned}
(I, V, W) >_{lex}^{term} (I', V', W') :\Leftrightarrow \quad & I \setminus \uparrow V \supset I' \setminus \uparrow V' \\
& \vee (I \setminus \uparrow V = I' \setminus \uparrow V' \\
& \qquad \wedge (V, W) >_{lex}^{bbr} (V', W')) ,
\end{aligned}
$$

where $(V,W) >_{lex}^{bbr} (V',W')$ is defined in Def. 3.1 on p. 63 as

$$(V,W) >_{lex}^{bbr} (V',W') \Leftrightarrow {\uparrow} V \subset {\uparrow} V' \vee ({\uparrow} V = {\uparrow} V' \wedge W \supset W')\,. \qquad \Diamond$$

Lemma 4.14. The lexicographical order for termination of Alg. 3.7 on p. 80, $>_{lex}^{term}$, is well-founded on triplets of finite sets.

Proof. We prove the well-foundedness of $>_{lex}^{term}$ on triplets (I,V,W) of finite sets in two steps:

1. The difference ${\downarrow} I \setminus {\uparrow} V$ is finite if I is finite. The proper superset relation $\supset$ is well-founded on finite sets as there exists no infinite strictly decreasing sequence $X_1 \supset X_2 \supset \cdots$ of finite sets $X_1, X_2, \ldots$.

2. From Lemma 3.8 on p. 64 we know that $>_{lex}^{bbr}$ is well-founded on pairs of finite sets.

As both relations used by $>_{lex}^{term}$ are well-founded on the sets they are applied to, the lexicographical order itself is well-founded on triplets of finite sets.
This concludes the proof. □

Abbreviations. For the termination proof of our framework, we define the following abbreviations. To ease readability, we recall these abbreviations in comments at the top of the proof outlines where they are used.
The first abbreviation $Inv_{term}(V,W)$ states that sets I, V, and W are finite and furthermore that sets W and ${\uparrow} V$ are disjoint—while the disjointness of these sets was introduced on intention (cf. Extension II of Sect. 3.2 on p. 69), it was neither necessary for the proof of partial correctness nor was it established as an invariant before. In our termination proof, however, we verify the claim of disjointness and exploit it.

$$Inv_{term}(V,W) := |V| \in \mathbb{N} \wedge |I| \in \mathbb{N} \wedge |W| \in \mathbb{N} \wedge W \cap {\uparrow} V = \emptyset \qquad (4.5)$$

The second abbreviation expresses that if W is empty, i.e. the framework's loop terminates, then the set of initial states and set ${\uparrow} V$ are disjoint. The idea behind this formula is simple: As soon as some path backwards from the final states ${\uparrow} F$ to the initial states ${\downarrow} I$ is found, set

W is emptied and only then the corresponding state is contained in $\uparrow V$. Unless such a path is found, $\uparrow V$ does not contain any initial state.

$$Inv_{VW}(V, W) := (W \neq \emptyset \Rightarrow I \cap \uparrow V = \emptyset) \tag{4.6}$$

In conjunction, these formulas are a loop invariant of the framework and we use them in the proof outline. As for partial correctness, the proof outline for termination is divided into several fragments as described in Alg. 4.1. The proof outline in Fig. 4.8 presents the invariant and termination term, and states pre- and postconditions for every placeholder which we will discuss in the rest of this section. All in all, it shows that the framework terminates if the input sets I and F are finite.
Throughout the proof outlines used to show termination, the value of variable T is of no interest. Therefore, the consequence rule (cf. Def. 2.21 on p. 44 or rather Def. 2.24 on p. 47) is applied after each assignment to T.

Overview. In contrast to the one for partial correctness, the proof outline for termination, presented in Fig. 4.8, does not contain any two consecutive assertions for which (non-trivial) logical implication has to be shown—but for the last two lines where there is a trivial implication. As in the termination proof of the basic BR, the proof is not concerned with what the algorithm does in detail. This is reflected by the trivial postcondition **true**. In the rest of this section, we show that every subalgorithm satisfies the necessary postcondition in the sense of total correctness. Termination of the complete algorithm is captured in Proposition 4.21 on p. 122.

Termination of the Subalgorithms. Correctness of the first subalgorithm INIT used in the proof outline is captured in the following lemma which describes that the initialisation of the algorithm's variables yields a state in which the invariant $Inv_{term}(V, W) \wedge Inv_{VW}(V, W)$ holds. The important detail in the proof outline Fig. 4.9 for this lemma is that V—and therefore $\uparrow V$—is empty. We use the "domain" and the "finite" cases of function assignment axioms (cf. Lemma 4.3) for the effect of the statements $T(S) := \bot$ and $T(W) := \varepsilon$.

// $Inv_{term}(V,W) := |V| \in \mathbb{N} \wedge |I| \in \mathbb{N} \wedge |W| \in \mathbb{N} \wedge W \cap \uparrow V = \emptyset$
// $Inv_{VW}(V,W) := (W \neq \emptyset \Rightarrow I \cap \uparrow V = \emptyset)$

1 $\{\, |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \,\}$
2 INIT; `/* see Lemma 4.15 */`
3 $\{\, |I| \in \mathbb{N} \wedge |V| \in \mathbb{N} \wedge |W| \in \mathbb{N} \wedge W \cap \uparrow V = \emptyset$
4 $\wedge\, (W \neq \emptyset \Rightarrow I \cap \uparrow V = \emptyset) \,\}$
5 $\{\,$ **inv:** $Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \,\}$ $\{\,$ **bd:** $(I,V,W) \,\}$
6 **while** $W \neq \emptyset$ **do**
7 $\alpha := (I,V,W)$;
8 $\{\, W \neq \emptyset \wedge Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \wedge (I,V,W) = \alpha \,\}$
9 NEXT_STATE; `/* see Lemma 4.16 */`
10 $\{\, x \notin \uparrow V \wedge x \notin W \wedge Inv_{term}(V, W \cup \{\, x \,\})$
11 $\wedge\, Inv_{VW}(V, W \cup \{\, x \,\}) \wedge (I, V, W \cup \{\, x \,\}) = \alpha \,\}$
12 **if** $x \in \downarrow I$ **then**
13 $\{\, x \in \downarrow I \wedge x \notin \uparrow V \wedge x \notin W \wedge Inv_{term}(V, W \cup \{\, x \,\})$
14 $\wedge\, Inv_{VW}(V, W \cup \{\, x \,\}) \wedge (I, V, W \cup \{\, x \,\}) = \alpha \,\}$
15 FOUND_TRACE `/* see Lemma 4.17 */`
16 $\{\, Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \wedge (I,V,W) <^{term}_{lex} \alpha \,\}$
17 **else**
18 $\{\, x \notin \downarrow I \wedge x \notin \uparrow V \wedge x \notin W \wedge Inv_{term}(V, W \cup \{\, x \,\})$
19 $\wedge\, Inv_{VW}(V, W \cup \{\, x \,\}) \wedge (I, V, W \cup \{\, x \,\}) = \alpha \,\}$
20 ADD_PREDECESSORS `/* see Lemma 4.18 */`
21 $\{\, Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \wedge (I,V,W) <^{term}_{lex} \alpha \,\}$
22 **fi**
23 $\{\, Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \wedge (I,V,W) <^{term}_{lex} \alpha \,\}$
24 **od**
25 $\{\, W = \emptyset \wedge Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \,\}$
26 $\{\,$ **true** $\,\}$

Figure 4.8.: Proof outline for termination of Alg. 3.7 on p. 80.

Lemma 4.15 (init and Termination). The correctness formula

$$\{\, |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \,\}$$
INIT
$$\{\, |I| \in \mathbb{N} \wedge |V| \in \mathbb{N} \wedge |W| \in \mathbb{N} \wedge W \cap {\uparrow} V = \emptyset \wedge (W \neq \emptyset \Rightarrow I \cap {\uparrow} V = \emptyset) \,\}$$

is true in the sense of total correctness.

Proof. By the proof outline in Fig. 4.9, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

```
 1   { |I| ∈ ℕ ∧ |F| ∈ ℕ }
 2  W := minimize(F);
 3   { W ⊆ F ∧ ↑W = ↑F ∧ |I| ∈ ℕ ∧ |F| ∈ ℕ }
 4   { |I| ∈ ℕ ∧ |W| ∈ ℕ }
 5  T(S) := ⊥;
 6   { (∀z ∈ S : T(z) = ⊥) ∧ |I| ∈ ℕ ∧ |W| ∈ ℕ }
 7   { |I| ∈ ℕ ∧ |W| ∈ ℕ }
 8  T(W) := ε;
 9   { (∀x ∈ W : T(x) ≠ ⊥) ∧ |I| ∈ ℕ ∧ |W| ∈ ℕ }
10   { |I| ∈ ℕ ∧ |W| ∈ ℕ }
11  V := ∅;
12   { V = ∅ ∧ |I| ∈ ℕ ∧ |W| ∈ ℕ ∧ W ∩ ↑V = ∅ }
13   { |I| ∈ ℕ ∧ |V| ∈ ℕ ∧ |W| ∈ ℕ ∧ W ∩ ↑V = ∅
14     ∧ (W ≠ ∅ ⇒ I ∩ ↑V = ∅) }
```

Figure 4.9.: Proof outline for termination – INIT.

Every iteration of the algorithmic framework's main loop begins with the selection of a state in W to be processed. The trace from x to the UCS of final states is stored in some temporary variable and the state x is then removed from W. In the following lemma we show that after the execution of code NEXT_STATE, x is guaranteed to be neither in ${\uparrow} V$ nor in W—because x was in W which is disjoint from ${\uparrow} V$ as Inv_{term} tells us—and we describe the effect on the loop invariant. For the termination

of this subalgorithm, we also employ the "singleton" special case of the function assignment axiom (cf. Corollary 4.4) for the effect of $T(x) := \bot$.

Lemma 4.16 (next_state and Termination). The correctness formula

$$\begin{aligned} &\{\, W \neq \emptyset \wedge Inv_{term}(V,W) \wedge Inv_{VW}(V,W) \wedge (I,V,W) = \alpha \,\} \\ &\textsc{next_state} \\ &\{\, x \notin \uparrow V \wedge x \notin W \wedge Inv_{term}(V, W \cup \{\, x \,\}\,) \\ &\quad \wedge Inv_{VW}(V, W \cup \{\, x \,\}\,) \wedge (I, V, W \cup \{\, x \,\}\,) = \alpha \,\} \end{aligned}$$

is true in the sense of total correctness.

Proof. By the proof outline in Fig. 4.10, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

In case the condition $x \in \downarrow I$ holds, a trace from $\uparrow F$ to the initial states has been found and a shortcut is taken by setting W to be the empty set, i.e. the algorithm terminates. The following lemma about FOUND_-TRACE states that the invariant holds after the shortcut code has been executed and, equally important, that the tuple (I, V, W) indeed decreased w.r.t. $<_{lex}^{term}$. The corresponding proof outline Fig. 4.11 uses that $Inv_{VW}(V, W \cup \{\, x \,\}\,)$ implies that $\uparrow V$ and I are disjoint as $W \cup \{\, x \,\}$ is non-empty. In the proof outline, we are not interested in the relationship between (I, V, W) and the value of variable α, regarding V and W, so we simply state that I did not change and α is (I, V', W'), where we use some temporary variables V', W'. While we introduce these temporary variables via existential quantification, their values do not change between assertions due to the equality $(I, V', W') = \alpha$ and α being fixed. With the previous observation we can deduce that $I \cap \uparrow V' = \emptyset$ holds. Notice that $x \in \downarrow I$ implies $I \cap \uparrow x \neq \emptyset$ as there is some $x' \in I$, s.t. $x \preceq x'$ holds and that this x' is also element of $\uparrow x$. Therefore, as V is set to $\{\, x \,\}$, the intersection of I and $\uparrow V$ is non-empty and we can conclude that $I \setminus \uparrow V$ is a proper subset of $I \setminus \uparrow V'$. In turn, this means that the tuple (I, V, W) is strictly less than $(I, V', W') = \alpha$ w.r.t. $<_{lex}^{term}$. This property is maintained until the end of FOUND_TRACE. As W is set to $\emptyset$, the invariant $Inv_{term}(V, W) \wedge Inv_{VW}(V, W)$ holds as well. We use the "finite" case of function assignment axioms (cf. Def. 3.3 on p. 76) and its

$$
\begin{array}{rl}
& /\!/ \ \mathit{Inv}_{term}(V,W) := |V| \in \mathbb{N} \wedge |I| \in \mathbb{N} \wedge |W| \in \mathbb{N} \wedge W \cap \uparrow V = \emptyset \\
& /\!/ \ \mathit{Inv}_{VW}(V,W) := (W \neq \emptyset \Rightarrow I \cap \uparrow V = \emptyset) \\
1 & \{ W \neq \emptyset \wedge \mathit{Inv}_{term}(V,W) \wedge \mathit{Inv}_{VW}(V,W) \wedge (I,V,W) = \alpha \} \\
2 & x := \mathit{select}(W); \\
3 & \{ x \in W \wedge W \neq \emptyset \wedge \mathit{Inv}_{term}(V,W) \\
4 & \ \wedge \mathit{Inv}_{VW}(V,W) \wedge (I,V,W) = \alpha \} \\
5 & \{ x \notin \uparrow V \wedge \mathit{Inv}_{term}(V,W) \wedge \mathit{Inv}_{VW} \wedge (I,V,W) = \alpha \} \\
6 & \tau := T(x); \\
7 & \{ \tau = T(x) \wedge x \notin \uparrow V \wedge \mathit{Inv}_{term}(V,W) \\
8 & \ \wedge \mathit{Inv}_{VW}(V,W) \wedge (I,V,W) = \alpha \} \\
9 & \{ x \notin \uparrow V \wedge \mathit{Inv}_{term}(V,W) \wedge \mathit{Inv}_{VW}(V,W) \wedge (I,V,W) = \alpha \} \\
10 & W := W \setminus \{ x \}; \\
11 & \{ x \notin \uparrow V \wedge x \notin W \wedge \mathit{Inv}_{term}(V,W) \cup \{ x \}) \\
12 & \ \wedge \mathit{Inv}_{VW}(V, W \cup \{ x \}) \wedge (I,V,W \cup \{ x \}) = \alpha \} \\
13 & T(x) := \bot; \\
14 & \{ T(x) = \bot \wedge x \notin \uparrow V \wedge x \notin W \wedge \mathit{Inv}_{term}(V, W \cup \{ x \}) \\
15 & \ \wedge \mathit{Inv}_{VW}(V, W \cup \{ x \}) \wedge (I,V,W \cup \{ x \}) = \alpha \} \\
16 & \{ x \notin \uparrow V \wedge x \notin W \wedge \mathit{Inv}_{term}(V, W \cup \{ x \}) \\
17 & \ \wedge \mathit{Inv}_{VW}(V, W \cup \{ x \}) \wedge (I,V,W \cup \{ x \}) = \alpha \}
\end{array}
$$

Figure 4.10.: Proof outline for termination – NEXT_STATE.

"singleton" special case (cf. Corollary 4.4) for the effect of the statements $T(W) := \bot$ and $T(x) := \tau$.

Lemma 4.17 (found_trace and Termination). The correctness formula

$$
\begin{array}{l}
\{ x \in \downarrow I \wedge x \notin \uparrow V \wedge x \notin W \wedge \mathit{Inv}_{term}(V, W \cup \{ x \}) \\
\ \wedge \mathit{Inv}_{VW}(V, W \cup \{ x \}) \wedge (I,V,W \cup \{ x \}) = \alpha \} \\
\text{FOUND_TRACE} \\
\{ \mathit{Inv}_{term}(V,W) \wedge \mathit{Inv}_{VW}(V,W) \wedge (I,V,W) <_{lex}^{term} \alpha \}
\end{array}
$$

is true in the sense of total correctness.

Proof. By the proof outline in Fig. 4.11, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

```
// Inv_term(V,W) := |V| ∈ ℕ ∧ |I| ∈ ℕ ∧ |W| ∈ ℕ ∧ W ∩ ↑V = ∅
// Inv_VW(V,W) := (W ≠ ∅ ⇒ I ∩ ↑V = ∅)
```

1 $\{\, x \in {\downarrow} I \wedge x \notin {\uparrow} V \wedge x \notin W \wedge Inv_{term}(V, W \cup \{\, x \,\}\,)$
2 $\wedge\, Inv_{VW}(V, W \cup \{\, x \,\}\,) \wedge (I, V, W \cup \{\, x \,\}\,) = \alpha \,\}$
3 $\{\, \exists V', W' : V' = V \wedge W' = W \cup \{x\} \wedge x \in {\downarrow} I \wedge x \notin {\uparrow} V$
4 $\wedge\, x \notin W \wedge Inv_{term}(V, W) \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
5 $\{\, \exists V', W' : x \in {\downarrow} I \wedge Inv_{term}(V, W)$
6 $\wedge\, x \notin V' \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
7 $V := \{\, x \,\}\,;$
8 $\{\, \exists V', W' : V = \{\, x \} \wedge x \in {\downarrow} I \wedge Inv_{term}(V, W)$
9 $\wedge\, x \notin V' \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
10 $\{\, \exists V', W' : I \setminus {\uparrow} V \subset I \setminus {\uparrow} V'$
11 $\wedge\, Inv_{term}(V, W) \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
12 $T(W) := \bot;$
13 $\{\, \exists V', W' : (\forall z \in W : T(z) = \bot) \wedge I \setminus {\uparrow} V \subset I \setminus {\uparrow} V'$
14 $\wedge\, Inv_{term}(V, W) \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
15 $\{\, \exists V', W' : I \setminus {\uparrow} V \subset I \setminus {\uparrow} V'$
16 $\wedge\, Inv_{term}(V, W) \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
17 $T(x) := \tau;$
18 $\{\, \exists V', W' : T(x) = \tau \wedge I \setminus {\uparrow} V \subset I \setminus {\uparrow} V'$
19 $\wedge\, Inv_{term}(V, W) \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
20 $\{\, \exists V', W' : I \setminus {\uparrow} V \subset I \setminus {\uparrow} V'$
21 $\wedge\, Inv_{term}(V, W) \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
22 $W := \emptyset$
23 $\{\, \exists V', W' : W = \emptyset \wedge I \setminus {\uparrow} V \subset I \setminus {\uparrow} V'$
24 $\wedge\, Inv_{term}(V, W) \wedge W' \neq \emptyset \wedge Inv_{VW}(V', W') \wedge (I, V', W') = \alpha \,\}$
25 $\{\, Inv_{term}(V, W) \wedge Inv_{VW}(V, W) \wedge (I, V, W) <^{term}_{lex} \alpha \,\}$

Figure 4.11.: Proof outline for termination – FOUND_TRACE.

Now we turn to the case when no trace has yet been found. Of course, this is more involved than the shortcut and we therefore follow the top-down approach where we dissect ADD_PREDECESSORS into further subalgorithms.

Since ADD_PREDECESSORS contains a loop itself, it is required to give a term that takes values in some well-founded domain. The loop under consideration simply empties a finite set O element by element and we

therefore choose O to be the termination term. The fact that O actually is finite stems from the finiteness constraint **Fin** of the optimized predecessor function (cf. Def. 3.7 on p. 83) to which it is set by $O := opb(x)$. The proper subset relation is well-founded on finite sets, so that we can deduce that this inner loop terminates if the subalgorithm PROCESS_-NEW_STATE does not interfere. As we will show in Lemma 4.19, this subalgorithm behaves well and does neither mess with O, nor does it break any other property necessary for our termination proof.

In a more detailed look into Lemma 4.18, which forms the main step in our termination proof, we first observe that the state under consideration x is neither in $\downarrow I$, nor in $\uparrow V$. Due to the properties of the minimization function (Def. 3.4 on p. 79), the value of $\mathit{minimize}(V \cup \{ x \})$ is a subset of $V \cup \{ x \}$ and a finite basis of $\uparrow(V \cup \{ x \})$ but we do not know its concrete elements. We therefore introduce a helper variable V' which has the old value of V, s.t. $(I, V, W \cup \{ x \}) = \alpha$.[1] Remember that α is the variable which stores the value of the termination term (I, V, W) from the beginning of the current iteration of the main loop. After the assignment $V := \mathit{minimize}(V \cup \{ x \})$, it holds that V is both a subset of $V' \cup \{ x \}$ and a finite basis of $\uparrow(V' \cup \{ x \})$. From the precondition we know that before the update to V, x was not in $\uparrow V$ which implies that $x \notin \uparrow V'$ holds and furthermore that $\uparrow V'$ is a proper subset of $\uparrow V$ which in turn means that $(I, V, W) <^{term}_{lex} (I, V', W \cup \{ x \})$ holds, if $I \setminus \uparrow V'$ is a subset of or equal to $I \setminus \uparrow V$.[2] As x is not in $\downarrow I$, we know that $I \setminus \uparrow V'$ and $I \setminus \uparrow V$ are the same and that $(I, V, W) <^{term}_{lex} (I, V', W \cup \{ x \})$ holds in fact. In the proof outline, we carry two conjuncts that imply $(I, V, W) <^{term}_{lex} (I, V', W \cup \{ x \})$: (1) $\uparrow V' \subset \uparrow V$ and (2) $\mathit{Inv}_{VW}(V, W \cup \{ x \})$ which states that I and $\uparrow V$ are disjoint as $W \cup \{ x \}$ is non-empty. In conjunction, we get that I and $\uparrow V'$ are also disjoint and that $(I, V, W) <^{term}_{lex} (I, V', W \cup \{ x \})$ is implied indeed. For the proof that PROCESS_NEW_STATE behaves well, we turn to Lemma 4.19.

Lemma 4.18 (add_predecessors and Termination). The correct-

[1] Note that α being fixed forces the existential quantifier we use to introduce the helper variable to always choose the same value.

[2] In this case, the lexicographical ordering is independent from the third component, $W \cup \{ x \}$. $(I, V, W) <^{term}_{lex} (I, V', X)$ holds for any finite set X.

ness formula

$$\begin{array}{l} \{\, x \notin \downarrow I \wedge x \notin \uparrow V \wedge x \notin W \wedge \mathit{Inv}_{term}(V, W \cup \{\, x \,\}) \\ \quad \wedge\ \mathit{Inv}_{VW}(V, W \cup \{\, x \,\}) \wedge (I, V, W \cup \{\, x \,\}) = \alpha \,\} \\ \textsc{add_predecessors} \\ \{\, \mathit{Inv}_{term}(V, W) \wedge \mathit{Inv}_{VW}(V, W) \wedge (I, V, W) <_{lex}^{term} \alpha \,\} \end{array}$$

is true in the sense of total correctness.

Proof. By the proof outline in Fig. 4.12, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

Subalgorithm PROCESS_NEW_STATE tests whether the optimized predecessor state y is new in the sense that it is not contained in $\uparrow V$ and, in the positive case, updates the set of states to be processed, as well as the stored trace for y. If state y is not new, it is discarded and no update is performed. The following Lemma 4.19 contains the corresponding proof outline where most assertions are mechanical but for the two following assignment $W := \mathit{minimize}(W \cup \{\, y \,\})$. There, it is crucial to accept that $\mathit{Inv}_{term}(V, W)$ holds for the updated set W as it is still disjoint from $\uparrow V$ due to the fact that y is not in $\uparrow V$.

Lemma 4.19 (process_new_state and Termination). The correctness formula

$$\begin{array}{l} \{\, \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset \wedge \uparrow V' \subset \uparrow V \\ \quad \wedge\ \mathit{Inv}_{term}(V, W) \wedge (I, V', W \cup \{\, x \,\}) = \alpha \,\} \\ \textsc{process_new_state} \\ \{\, \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset \wedge \uparrow V' \subset \uparrow V \\ \quad \wedge\ \mathit{Inv}_{term}(V, W) \wedge (I, V', W \cup \{\, x \,\}) = \alpha \,\} \end{array}$$

is true in the sense of total correctness.

Proof. By the proof outline in Fig. 4.13, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

// $Inv_{term}(V,W) := |V| \in \mathbb{N} \wedge |I| \in \mathbb{N} \wedge |W| \in \mathbb{N} \wedge W \cap \uparrow V = \emptyset$
// $Inv_{VW}(V,W) := (W \neq \emptyset \Rightarrow I \cap \uparrow V = \emptyset)$

1 $\{ x \notin \downarrow I \wedge x \notin \uparrow V \wedge x \notin W \wedge Inv_{term}(V, W \cup \{x\})$
2 $\wedge\ Inv_{VW}(V, W \cup \{x\}) \wedge (I, V, W \cup \{x\}) = \alpha \}$
3 $\{ \exists V' : V' = V \wedge I \cap \uparrow(V' \cup \{x\}) = \emptyset \wedge x \notin \uparrow V' \wedge x \notin W$
4 $\wedge\ Inv_{term}(V', W \cup \{x\}) \wedge (I, V', W \cup \{x\}) = \alpha \}$
5 $V := minimize(V \cup \{x\});$
6 $\{ \exists V' : \uparrow(V' \cup \{x\}) = \uparrow V \wedge I \cap \uparrow V = \emptyset \wedge x \notin \uparrow V' \wedge x \notin W$
7 $\wedge\ Inv_{term}(V', W \cup \{x\}) \wedge (I, V', W \cup \{x\}) = \alpha \}$
8 $\{ \exists V' : I \cap \uparrow V = \emptyset \wedge \uparrow V' \subset \uparrow V$
9 $\wedge\ Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
10 $O := opb(x);$
11 $\{ \exists V' : |O| = opb(x) \wedge I \cap \uparrow V = \emptyset \wedge \uparrow V' \subset \uparrow V$
12 $\wedge\ Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
13 $\{ \exists V' : |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset \wedge \uparrow V' \subset \uparrow V$
14 $\wedge\ Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
15 $\{$ **inv:** $\exists V' : |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset \wedge \uparrow V' \subset \uparrow V$
16 $\wedge\ Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$ $\{$ **bd:** $O \}$
17 **while** $O \neq \emptyset$ **do**
18 $\beta := O;$
19 $\{ \exists V' : O \neq \emptyset \wedge O = \beta \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset$
20 $\wedge\ \uparrow V' \subset \uparrow V \wedge Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
21 **let** $y \in O;$
22 $\{ \exists V' : y \in O \wedge O = \beta \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset$
23 $\wedge\ \uparrow V' \subset \uparrow V \wedge Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
24 $O := O \setminus \{y\};$
25 $\{ \exists V' : y \notin O \wedge O \cup \{y\} = \beta \wedge |O \cup \{y\}| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset$
26 $\wedge\ \uparrow V' \subset \uparrow V \wedge Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
27 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset$
28 $\wedge\ \uparrow V' \subset \uparrow V \wedge Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
29 PROCESS_NEW_STATE `/* see Lemma 4.19 */`
30 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset$
31 $\wedge\ \uparrow V' \subset \uparrow V \wedge Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
32 **od**
33 $\{ \exists V' : O = \emptyset \wedge |O| \in \mathbb{N} \wedge I \cap \uparrow V = \emptyset$
34 $\wedge\ \uparrow V' \subset \uparrow V \wedge Inv_{term}(V, W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
35 $\{ Inv_{term}(V, W) \wedge Inv_{VW}(V, W) \wedge (I, V, W) <^{term}_{lex} \alpha \}$

Figure 4.12.: Proof outline for termination – ADD_PREDECESSORS.

```
// Inv_term(V,W) := |V| ∈ ℕ ∧ |I| ∈ ℕ ∧ |W| ∈ ℕ ∧ W ∩ ↑V = ∅
// Inv_VW(V,W) := (W ≠ ∅ ⇒ I ∩ ↑V = ∅)
```

1 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
2 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
3 **if** $y \notin {\uparrow} V$ **then**
4 $\{ \exists V' : y \notin {\uparrow} V \wedge O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
5 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
6 $W := minimize(W \cup \{y\});$
7 $\{ \exists V' : y \in {\uparrow} W \wedge y \notin {\uparrow} V \wedge O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
8 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
9 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
10 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
11 $\sigma := wit_x(y) \cdot \tau;$
12 $\{ \exists V' : \sigma = wit_x(y) \cdot \tau \wedge O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
13 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
14 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
15 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
16 UPDATE__TRACE `/* see Lemma 4.20 */`
17 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
18 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
19 **else**
20 $\{ \exists V' : y \in {\uparrow} V \wedge O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
21 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
22 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
23 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
24 **skip**
25 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
26 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$
27 **fi**
28 $\{ \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset$
29 $\wedge {\uparrow} V' \subset {\uparrow} V \wedge Inv_{term}(V,W) \wedge (I, V', W \cup \{x\}) = \alpha \}$

Figure 4.13.: Proof outline for termination – PROCESS__NEW__STATE.

The remaining Lemma 4.20 uses that UPDATE_TRACE only changes T—which does not occur in its pre- and postcondition—and nothing else. Here, every assertion is mechanical, i.e. the property

$$\{\, \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset \wedge {\uparrow} V' \subset {\uparrow} V \\ \wedge\, Inv_{term}(V, W) \wedge (I, V', W \cup \{\, x \,\}) = \alpha \,\}$$

is conserved throughout the execution of the subalgorithm and the conjuncts introduced by statements are discarded. We employ the "singleton" special case of the function assignment axiom (cf. Corollary 4.4) for the effect of $T(y) := \sigma$.

Lemma 4.20 (update_trace and Termination). The correctness formula

$$\{\, \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset \wedge {\uparrow} V' \subset {\uparrow} V \\ \wedge\, Inv_{term}(V, W) \wedge (I, V', W \cup \{\, x \,\}) = \alpha \,\}$$

UPDATE__TRACE

$$\{\, \exists V' : O \subset \beta \wedge |O| \in \mathbb{N} \wedge I \cap {\uparrow} V = \emptyset \wedge {\uparrow} V' \subset {\uparrow} V \\ \wedge\, Inv_{term}(V, W) \wedge (I, V', W \cup \{\, x \,\}) = \alpha \,\}$$

is true in the sense of total correctness.

Proof. By the proof outline in Fig. 4.14, Lemma 2.12 on p. 48 (proof outlines imply correctness theorems) and Lemma 2.11 on p. 46 (soundness of PW and TW). □

Employing the lemmas 4.15 through 4.20, we accept the validity of the proof outline shown in Fig. 4.8 which we presented at the beginning of this section and conclude with the following proposition.

Proposition 4.21 (Termination of the Algorithmic Framework). Given a WSLTS $(S, L, \rightarrow, \preceq)$ with a decidable $\preceq$ and an effective pred-basis, our algorithmic framework terminates for any finite sets $I, F \subseteq S$ if *select*, *opb*, *wit* are adequately instantiated. Formally,

$$\vdash_{TW} \{\, |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \,\}\ \text{ALGORITHMIC FRAMEWORK}\ \{\, \mathbf{true} \,\}\ .$$

Proof. By the proof outline in Fig. 4.8 on p. 113 together with Lemma 4.15 (INIT), Lemma 4.16 (NEXT_STATE), Lemma 4.17 (FOUND_TRACE), Lemma 4.18 (ADD_PREDECESSORS), Lemma 4.19

```
   // Inv_term(V,W) := |V| ∈ ℕ ∧ |I| ∈ ℕ ∧ |W| ∈ ℕ ∧ W ∩ ↑V = ∅
   // Inv_VW(V,W) := (W ≠ ∅ ⇒ I ∩ ↑V = ∅)
 1   { ∃V′ : O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅ ∧ ↑V′ ⊂ ↑V
 2     ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
 3 if T(y) = ⊥ ∨ |T(y)| > |σ| then
 4       { ∃V′ : (T(y) = ⊥ ∨ |T(y)| > |σ|)
 5         ∧ O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅
 6         ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
 7     T(y) := σ
 8       { ∃V′ : T(y) = σ ∧ O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅
 9         ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
10       { ∃V′ : O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅
11         ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
12 else
13       { ∃V′ : ¬(T(y) = ⊥ ∨ |T(y)| > |σ|) ∧ O ⊂ β ∧ |O| ∈ ℕ
14         ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
15       { ∃V′ : O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅
16         ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
17     skip
18       { ∃V′ : O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅
19         ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
20 fi
21   { ∃V′ : O ⊂ β ∧ |O| ∈ ℕ ∧ I ∩ ↑V = ∅
22     ∧ ↑V′ ⊂ ↑V ∧ Inv_term(V,W) ∧ (I, V′, W ∪ { x }) = α }
```

Figure 4.14.: Proof outline for termination – UPDATE_TRACE.

(PROCESS_NEW_STATE), Lemma 4.20 (UPDATE_TRACE) and Lemma 2.12 on p. 48 (proof outlines imply correctness theorems). ☐

4.4. Total Correctness

In the previous sections we have shown that each adequate instantiation of the framework terminates for any valid input that is a labelled coverability problem and that if it terminates its result is a correct answer to that problem. The decomposition lemma allows to combine both partial correctness and termination to conclude total correctness of our algorithmic framework.

Theorem 4.22. Consider WSLTS $(S, L, \rightarrow, \preceq)$ with decidable $\preceq$ and effective pred-basis and let *select*, *opb*, *wit* be adequately instantiated. If the algorithmic framework is run with finite sets $I, F \subseteq S$ as input, it terminates with a finite result V, T, s.t. if and only if F is coverable from I, i.e. $I \hookrightarrow F$, then $\uparrow V \cap \downarrow I$ is non-empty. For each state x in the intersection of V and $\downarrow I$, $T(x)$ is a trace from x to $\uparrow F$. Formally,

$$\begin{aligned} \models_{tot} \quad & \{\, |I| \in \mathbb{N} \wedge |F| \in \mathbb{N} \,\} \\ & \textsc{Algorithmic Framework} \\ & \Big\{\, (dist(\downarrow I, \uparrow F) \neq \infty \Leftrightarrow \uparrow V \cap \downarrow I \neq \emptyset) \\ & \quad \wedge \forall x \in V \cap \downarrow I : x \xhookrightarrow{T(x)} F \,\Big\}\,. \end{aligned}$$

Proof. By Corollary 3.13 on p. 82, the finiteness of $dist(\downarrow I, \uparrow F)$ is equivalent to F being coverable from I. The total correctness of the algorithmic framework (Alg. 3.7 on p. 80) follows from Proposition 4.13 (partial correctness of the algorithmic framework) and Proposition 4.21 (termination of the algorithmic framework), together with Def. 2.23 on p. 46 (decomposition rule). ☐

This closes the proof of our algorithmic framework.

4.5. Differences in the Partial Correctness Proofs

While the termination proofs for the basic BR and the algorithmic framework work alike, there is a greater difference between the proofs of partial correctness of the two algorithms. One point that factors into the difference between the two proofs are the types of problems they solve. The framework algorithm solves the *labelled* coverability problem and thus is concerned with sequences of transition labels whereas the basic algorithm does not have to keep track of such traces. More importantly, in the proof of the basic BR (Sect. 3.1.1 on p. 53), we had to use fundamental properties of the reflexive, transitive closure of the predecessor relation, the upward-closure operator and the definition of the predecessor basis (lemmas 3.3, 3.4, and 3.5). To some extent, these properties can be considered to be *low-level.* Contrastingly, the framework was proved under the application of the reasonably *high-level* properties **Dist** and **Wit** of the distance and witness functions, defined in Sect. 3.4 on p. 81. As the predecessor basis function *pb* is an instantiation of the optimized predecessor function *opb* (cf. Sect. 3.4.1 on p. 82), the powerful distance function and the **Dist** constraint are available if one has to show further properties of the basic algorithm.

PART II

Instantiating the Framework

CHAPTER FIVE

SSC Instances and Guided Search

I may not have gone where I intended to go, but I think I have ended up where I needed to be.
— Douglas Adams, *The Long Dark Tea-Time of the Soul*

Contents

5.1 Backward Acceleration – A Novel Approach 131
  5.1.1 Backward Acceleration is an SSC 136
  5.1.2 Practical Approach to Backward Acceleration 139
  5.1.3 Computation of Maximal Extensions for Petri Nets 141
5.2 Pruning 143
5.3 Partial-Order Reduction 147
5.4 Combination of Search Space Constructions 152
5.5 Guided Search 155
  5.5.1 Syntactic Distance and Syntactic Weight . . 156

In this chapter, we inspect several methods to increase the performance of backward reachability analysis and show that these are, in fact, instances of our concept of search space constructions by proving that they adhere to conditions **Dist** and **Wit**. These proofs are elegant, rather straightforward implications from the definitions of the methods which emphasizes the generality and accessibility of our framework to new optimizations. In Sect. 5.5, we discuss how the analysis can be improved by employing strategies to guide the search.

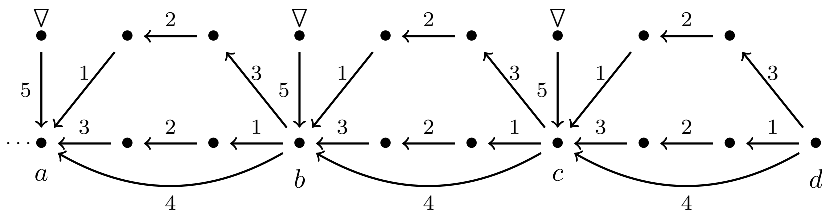
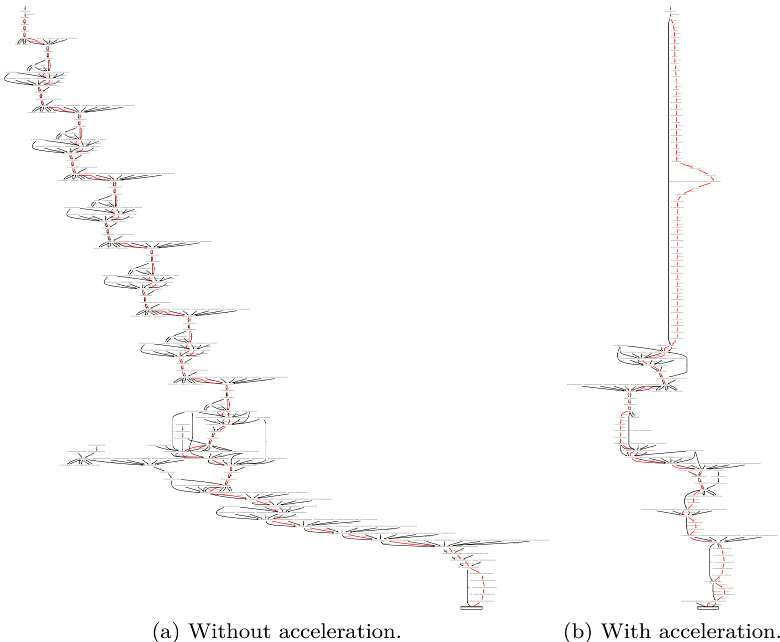


Figure 5.1.: Schematic example search space.

To visualize and compare the instances of search space constructions, we show their effect when applied to a schematic state space (as well as our running example Petri net N_{ex}). The state space displayed in Fig. 5.1 is explored backwards from state a. Transitions are labelled with numbers $1, \ldots, 5$ and four interesting states are labelled with a, b, c, d. The set of initial states is $I = \{ c \}$ and the final state is $F = \{ a \}$. The part to the left of a is extraneous to our examples. In this example, the WQO between states is irrelevant for unlabelled states, but for labelled states it is $a > b > c > d$. Triangles indicate large regions of the state space that have to be explored if the adjacent state is explored. When an SSC can avoid the exploration of certain states in the following examples, these are marked grey. Note that the portion of the state space to the right of initial state c usually is not explored. However, it is allowed for SSCs to *skip* over c and explore one of those states. Our acceleration may do so.

5.1. Backward Acceleration – A Novel Approach

When we inspected the backward search space for several case studies, we found regions that formed recurring patterns. The reason was the repetition of transition sequences. The idea of backward acceleration is to identify such repeating sequences. Intuitively, at the second occurrence of σ in $y \overset{\sigma}{\hookrightarrow} x \overset{\sigma}{\hookrightarrow} z$ with $y \prec x \prec z$ we compute the *maximal extension* of σ: the transition sequence $y' \overset{\tau}{\hookrightarrow} x$ with $\tau = \sigma^k$, k maximal. This means the acceleration computes, in one step, the effect of a maximal number of iterations of σ.



(a) Without acceleration. (b) With acceleration.

Figure 5.2.: Recurring patterns in the search space of the Kanban case study.

Where applicable, backward acceleration can drastically reduce the explored search space. Take for example the two graphs in Fig. 5.2 which represent different search spaces of the Kanban case study (which we recall in Sect. B.3 on p. 250), a P/T Petri net. Both graphs were automatically generated by our tool PETRUCHIO/BW and both show traces from the top to the bottom. Some marking, covered by the initial marking, is located at the top and the single target marking is located at the bottom. Black lines of the graph were explored backwards and the gray lines indicate the exact trace found in order to give evidence that the target marking indeed is coverable from the initial marking.

In Fig. 5.2a, a form of repetition is clearly visible—both in the graph's upper part with large "plateaus" and in its lower part. Our backward acceleration is capable of detecting some specific case of repetition and may then introduce a short-cut in the exploration of the search space as depicted in Fig. 5.2b. Here, after a single occurrence of the repeating "plateau"-pattern (center of graph), backward acceleration kicks in and delivers the maximal extension of the pattern in one step while still conserving information on the transition sequence. The long gray line on the right hints on how many exploration steps are omitted.[1]

We propose a two-step approach for the implementation of backward acceleration: Given some state x and $y \in pb(x)$, we construct a set of states $x' \in S$ so that y is backward reachable from x' and x' is greater than y. (In Fig. 5.3 it corresponds to $x = \circ$, $y = b$, and $x' = a$.) Transition sequences σ connecting x' and y, i.e. $y \overset{\sigma}{\hookrightarrow} x'$, are candidate sequences for extension to some maximal k s.t. state $y' \prec y$ is reached via $y' \overset{\sigma^k}{\hookrightarrow} y$. In this example, we call y', y and all the intermediate states $y_0, \ldots, y_k$ in the sequence

$$y' = y_k \overset{\sigma}{\hookrightarrow} y_{k-1} \overset{\sigma}{\hookrightarrow} \cdots \overset{\sigma}{\hookrightarrow} y_0 = y \overset{\sigma}{\hookrightarrow} x' \ ,$$

accelerated predecessors of x'. Formally, the set of (*acceleration candidate*) states from which candidate sequences might originate is

$$\{\, x' \in S \mid y \prec x' \wedge y \in pre^*(x') \,\} \ .$$

[1] The nose-like bend to the right stems from the automatic layout of the graph. The gray line goes around the (very long) inscription of the one long black arrow.

The definition of the backward acceleration SSC takes some subset of the accelerated predecessors and uses it in an conservative extension of the predecessor basis to determine function *opb*. The reason to only take a subset instead of all the accelerated predecessors is to allow for reasonable implementations of the optimization: As the set of acceleration candidates takes into account all states x', s.t. $y \prec x' \in suc^*(y)$, constructing this set itself poses a labelled coverability problem.[2] The weakened restriction of asking for a subset of acceleration candidates enables implementations to consider only those acceleration candidates that are "easy" to find, for example those in set V of our framework algorithm. We will go into more detail in Sect. 5.1.2 where we discuss our implementation.

Definition 5.1 (Backward Acceleration). Given a WSLTS $(S, L, \rightarrow, \preceq)$ and some state x. Consider a subset A_x of the set of all *accelerated predecessors* of x, i.e.

$$A_x \subseteq \{\, y' \in pre^*(\uparrow pb(x)) \mid \exists y \in pb(x), x' \in S, \sigma \in L^*, k \in \mathbb{N} : \\ y' \prec y \prec x' \wedge y' \overset{\sigma^k}{\hookrightarrow} y \overset{\sigma}{\hookrightarrow} x' \,\} \,.$$

The optimized predecessor function *backward acceleration* is defined as

$$opb_{accel}(x) := minimize(A_x \cup pb(x)) \,.$$

The corresponding witness function $wit_{accel,x}$ maps each $y' \in opb_{accel}(x)$ to either $lbl_x(y')$ if y' is a one-step predecessor of x or to the path $\sigma^k \cdot lbl_x(y)$, where σ and y are used in the computation of set A_x. ◊

This definition implies that for each acceleration $y' \overset{\sigma^k}{\hookrightarrow} y$ only those y' with maximal k are retained as optimized predecessors by the basis construction via *minimize*. Note that y is strictly larger than each of its accelerated states y' and is removed.

[2] We begin at initial states $I = \{\, y \,\}$ and ask the question if a finite basis of the set of states that are strictly greater than y, i.e. $F = \text{basis}(\{\, x' \in S \mid y \prec x' \,\})$—which exists due to Lemma 2.6 on p. 22 (finite basis) as the set of states greater than y is upward-closed—, is coverable (cf. Def. 2.16 on p. 35). If the answer is "yes", we have found an acceleration candidate x' and a corresponding trace σ with $y \overset{\sigma}{\hookrightarrow} x'$ as returned by algorithm Alg. 3.7 on p. 80.

Figure 5.3 illustrates the effect of backward acceleration. Assume a is in the basis of processed states V and we currently explore the unlabelled state $\circ$. In a first step, the algorithm follows transition 1 and finds state b. It discovers that a is strictly larger than b and that a is a successor of b. Sequence 3 2 1 leads backward from a to b, and is thus a candidate for backward acceleration. To compute its maximal extension, we follow 3 2 1 from b to state c. Since c again is strictly smaller than b, we repeat the procedure and reach $d \prec c$. From d the sequence does not lead to a smaller state, which terminates the acceleration. The set of accelerated states $\{ c, d \}$ is combined with the predecessor basis $pb(\circ)$. Minimization of the resulting set yields $\{ d \}$.

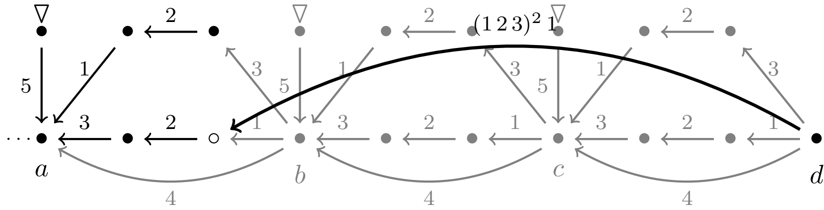


Figure 5.3.: Schematic example search space for acceleration.

Example 5.1 (Backward Acceleration for N_{ex}). In Fig. 5.4 we have drawn the search space of N_{ex} w.r.t. m_Ω where we used backward acceleration to reduce the number of states visited. States that are visited in the full search space (cf. Fig. 2.7 on p. 32) but not in the reduced space in this example are coloured gray.
The effect of the backward acceleration is clearly visible in the two long arrows labelled $(t_3t_1)^1t_3$ and $(t_3t_5t_1)^1t_3$ that originate in p_2 and point to $p_3 + 2p_5$ and $p_3 + p_5 + p_7$. To understand the process of acceleration starting from $p_3 + p_5 + p_7$, we observe the covering predecessor w.r.t. t_3, i.e. $p_2 + p_7$, which is strictly less than state $p_2 + 2p_7$ which the algorithm has already visited at this stage (it is the final state m_Ω). Backward acceleration tells us that $p_2 + 2p_7$ is an acceleration candidate and a candidate sequence is $\sigma = t_3t_5t_1$ as $p_2 + 2p_7 \overset{\sigma}{\hookrightarrow} p_2 + p_7$ holds. Notice that while there are several different candidate sequences that we can describe by the regular expression $t_3t_5t_1(t_3t_1)^* + t_3t_1t_5(t_3t_1)^*$ in this example we consider only one sequence.

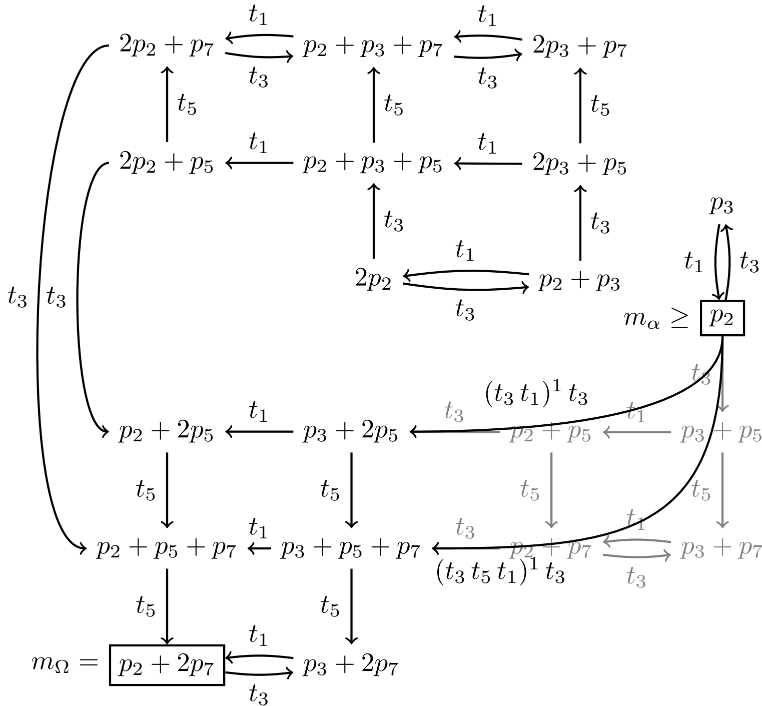


Figure 5.4.: Exploration of N_{ex}, m_Ω with backward acceleration.

In the second step of backward acceleration, the set of accelerated predecessors is constructed. Therefore, from the current state $p_2 + p_7$ the algorithm constructs the search space along the candidate sequence. Following the sequence once leads to state p_2 (as $p_2 \overset{\sigma}{\hookrightarrow} p_2 + p_7$) which is strictly less than $p_2 + p_7$. Taking the sequence a second time leads to $p_2 + p_5$. Since this state is not strictly less than p_2, backward acceleration does not consider $p_2 + p_5$ as part of the accelerated predecessors. No further iteration of the sequence leads to a lesser state and the set of accelerated predecessors is determined to be $\{ p_2 \}$.

The result of the backward acceleration is defined to be *minimize*$(\{ p_2 \} \cup pb(p_3 + p_5 + p_7))$ which is *minimize*$(\{ p_2 \} \cup \{ p_2 + p_7, p_3 + 2p_5 \})$. Mini-

mization leaves us with optimized predecessors $p_3 + 2p_5$ (via t_5) and the accelerated predecessor p_2 (via $(t_3 t_5 t_1)^1 t_3$).

To complete the picture, backward acceleration has a positive effect when exploring the pred-basis of p_3+2p_5. There, acceleration candidate p_2+2p_5 is found as a greater successor of $p_2 + p_5$ via trace $t_3 t_1$ which is then followed to lead to p_2.

A situation were backward acceleration has no positive effect is found at the top part of the graph. A covering predecessor of $p_2 + p_3 + p_5$ w.r.t. transition label t_3 is $2p_2$ which is strictly less than both $2p_2+p_7$ and $2p_2+p_5$. Nevertheless, attempting to follow the corresponding candidate sequences leads to $2p_2$—both via $t_3 t_1 t_5$ and $t_3 t_1$—which is not less than $2p_2$ and therefore is not considered an accelerated predecessor.

In comparison to the original search space, backward acceleration allowed the backward analysis to skip over four states. $\Diamond$

In the sections 5.2 and 5.3 we recall pruning and partial-order techniques which reduce the "out-degree" of the traversed state space, i.e. its "breadth". Contrasting to those techniques, backward acceleration can reduce its "depth". Another difference is that backward acceleration is applicable to WSLTSs in general, no notion of dependence of transitions (as needed for partial-order reduction) and no over-approximation of the system's state space (as for pruning) is required. This generality hints that backward acceleration may not perform as well as model-specific optimizations. We will discuss the performance in Sect. 7.2 on p. 207 where we present the results of our experimental evaluation. In Sect. 5.1.2 we discuss how the maximal extension can be computed in general. However, for certain WSLTSs the state resulting from the maximal extension σ^k of a candidate sequence σ might be computed directly without actually executing σ^k. In Sect. 5.1.3 we look at the computation of accelerated predecessors for PN markings.

5.1.1. Backward Acceleration is an SSC

As a high-level argument for backward acceleration being an SSC, consider the set

$$\uparrow opb_{accel}(x) = \uparrow minimize(A_x \cup pb(x)) = \uparrow A_x \cup \uparrow pb(x)$$

which subsumes the upward-closure of the pred-basis, i.e. $\uparrow pb(x) \subseteq \uparrow opb_{accel}(x)$. Since already $pb(x)$ is distance reducing, we conclude

$$dist(\downarrow I, \uparrow opb_{accel}(x)) \leq dist(\downarrow I, pb(x)) < dist(\downarrow I, \uparrow x)\,.$$

The full proof is captured in Proposition 5.1.

Proposition 5.1. Backward acceleration is a search space construction.

Proof. Let $\mathcal{S} = (S, L, \rightarrow, \preceq)$ a WSLTS, $V, I \subseteq S$ sets of states, $x \in S$ a state, and A_x a set of accelerated states w.r.t. x and $opb_{accel}(x) = minimize(A_x \cup pb(x))$.

- Due to the well-foundedness property of the WQO, implying that there are no infinite strictly decreasing sequences of states, set A_x is finite, i.e. condition **Fin** is satisfied. With the finiteness of $pb(x)$ and the finiteness of the result of the *minimize* function, $opb_{accel}(x)$ is finite.

- To show that condition **Dist** is met, we relate $pre^*(\uparrow opb_{accel}(x))$ with $pre^*(\uparrow A_x)$ and $pre^*(\uparrow pb(x))$.

$$\begin{aligned}
&pre^*(\uparrow opb_{accel}(x)) \\
&= pre^*(\uparrow minimize(A_x \cup pb(x))) \\
&\qquad\qquad\qquad\qquad \text{(backward acceleration [Def. 5.1])} \\
&= pre^*(\uparrow(A_x \cup pb(x))) \qquad \text{(minimize [Def. 3.4 on p. 79])} \\
&= pre^*(\uparrow A_x \cup \uparrow pb(x)) \qquad \text{(Distr. } \uparrow, \cup \text{ [Lemma 2.5-2 on p. 21])} \\
&= pre^*(\uparrow A_x) \cup pre^*(\uparrow pb(x)) \\
&\qquad\qquad\qquad\qquad \text{(Distr. } pre, \cup \text{ [Lemma 2.7-1 on p. 24])}
\end{aligned}$$

  To simplify this union, we show that $pre^*(\uparrow A_x)$ is a subset of

$pre^*(\uparrow pb(x))$.

$$
\begin{aligned}
A_x &\subseteq pre^*(\uparrow pb(x)) \quad \text{(backward acceleration [Def. 5.1])}\\
\Rightarrow \uparrow A_x &\subseteq \uparrow pre^*(\uparrow pb(x))\\
&\qquad \text{(Monotonicity } \uparrow \text{ [Lemma 2.5-3 on p. 21])}\\
\Rightarrow pre^*(\uparrow A_x) &\subseteq pre^*(\uparrow pre^*(\uparrow pb(x)))\\
&\qquad \text{(Monotonicity } pre^* \text{ [Lemma 2.7-2 on p. 24])}\\
\Rightarrow pre^*(\uparrow A_x) &\subseteq pre^*(pre^*(\uparrow pb(x)))\\
&\qquad (pre^* \text{ of a UCS [Lemma 2.8 on p. 26])}\\
\Rightarrow pre^*(\uparrow A_x) &\subseteq pre^*(\uparrow pb(x))\\
&\qquad \text{(Idempotence } pre^* \text{ [Lemma 2.7-3 on p. 25])}
\end{aligned}
$$

In conjunction, we conclude that

$$pre^*(\uparrow opb_{accel}(x)) = pre^*(\uparrow pb(x))$$

holds, expressing that the states backward reachable from the optimized (accelerated) predecessors of x are exactly those backward reachable from the one-step predecessors of x. Assume that $dist(\downarrow I, \uparrow x) \neq \infty$ and further that condition **Dist** is violated, i.e.

$$dist(\downarrow I, \uparrow opb_{accel}(x)) \geq dist(\downarrow I, \uparrow x)\,.$$

This implies that for every state $y \in pre^*(\uparrow opb_{accel}(x))$ the distance $dist(\downarrow I, y)$ is at least $dist(\downarrow I, \uparrow x)$. As we have seen, this is equivalent to the statement that for every state $y \in pre^*(\uparrow pb(x))$ the distance $dist(\downarrow I, y)$ is at least $dist(\downarrow I, \uparrow x)$, which implies

$$dist(\downarrow I, \uparrow pb(x)) \geq dist(\downarrow I, \uparrow x)\,.$$

This contradicts the definitions of pb and $dist$. Hence, condition **Dist** is satisfied by opb_{accel}.

- The $wit_{accel,x}$ function meets the **Wit** condition by definition.

We conclude that backward acceleration is a search space construction.

□

5.1.2. Practical Approach to Backward Acceleration

As stated in the context of Def. 5.1, the question whether a state can be accelerated, i.e. if there is a acceleration candidate and a candidate sequence, is a coverability problem in itself and we therefore allow to restrict the search for such candidates. In our reference implementation of this SSC, upon the inspection of some state $y \in pb(x)$ the basis of visited states V (cf. algorithmic framework Alg. 3.7 on p. 80) is searched for acceleration candidates $x' \succ y$. The framework knows the trace from y to the final states to be $lbl_y(x) \cdot \tau$, where τ is a trace from x to the upward-closure of final states, i.e. $y \xhookrightarrow{lbl_y(x)} x \xhookrightarrow{\tau} F$.

There are at least two methods to find a candidate sequence from x' to y and also to check if y indeed is a covering predecessor of x'.

1. Starting from y, follow along trace $lbl_y(x) \cdot \tau$ and keep track if one of the intermediate states is greater or equal to x'. For example, let $lbl_y(x) \cdot \tau$ be $t_1 t_2 \ldots t_n$ and compute $y \xhookrightarrow{t_1} s_1 \xhookrightarrow{t_2} s_2 \xhookrightarrow{t_3} \cdots \xhookrightarrow{t_n} s_n$. If $s_i \succeq x'$, then the prefix $t_1 t_2 \ldots t_i$ is a candidate sequence since x' is coverable from y via that sequence, i.e. $y \xhookrightarrow{t_1 t_2 \cdots t_i} x'$.

2. To retrieve a candidate trace σ from a state y and an acceleration candidate x', we may use that the algorithmic framework allows to lose traces of states not in W, but the correctness proof does not require to do so: Function T may associate $\bot$ with states in W, but if it associates a trace with state x', then the trace has to lead from that state to the upward-closure of the final states.

   Let γ_y be $lbl_y(x) \cdot \tau$ from above, s.t. $y \xhookrightarrow{\gamma_y} F$. If we have a trace $\gamma_{x'}$ from x' to $\uparrow F$, i.e. $x' \xhookrightarrow{\gamma_{x'}} F$, then we can easily identify if y is a covering predecessor of x' by testing if $\gamma_{x'}$ is a suffix of γ_y. Furthermore, the prefix of γ_y of length $|\gamma_y| - |\gamma_{x'}|$—where $|\gamma|$ is the length of sequence γ—is a candidate sequence. In Fig. 5.5 an example of this approach is given with transition labels $0, 1, \ldots, 7$.

   This is the option we chose for our reference implementation.

Computing the accelerated predecessors of y w.r.t. the candidate sequence σ is rather straight-forward. Consider the next accelerated pre-

$$\begin{array}{rcccccccc} \gamma_{x'} = & & & & & 6 & 7 & 6 & 5 \\ \gamma_y = & 0 & 1 & 2 & 3 & 6 & 7 & 6 & 5 \end{array}$$

candidate sequence — equal suffix

Figure 5.5.: Example of constructing a candidate sequence

decessor of y is y', as $y' \overset{\sigma}{\hookrightarrow} y$, and the sequence σ is $t_1 t_2 \dots t_n$. We construct the sets $Y_{1,n+1} = \{ y \}$ and

$$Y_{1,i} = \text{basis}(\left\{ z \in S \mid z \overset{t_i}{\hookrightarrow} Y_{1,i+1} \right\}) \text{ for each } n \geq i \geq 1\,,$$

s.t. $Y_{1,i}$ is the predecessor basis of the states in $Y_{1,i+1}$ w.r.t. transition label t_i. To compute the covering predecessors w.r.t. a specific transition label t, we can use the pred-basis of the respective states, for example $pb(y)$, and retain only those states $z \in pb(y)$ with $lbl_z(y) = t$.[3]

As the result, the set $Y_{1,1}$ is a basis of states from which y is coverable via σ^1. Thus, all states in $Y_{1,1}$ which are strictly less than y are accelerated predecessors of y. Starting from these accelerated predecessors, i.e. $Y_{2,n+1} = \{ y_1 \in Y_{1,1} \mid \exists z \in Y_{1,n+1} : y_1 \prec z \}$, we reiterate the process to construct a basis of states from which y is coverable via σ^2. Successively, we build the sequence of sets

$$\begin{aligned} Y_{j,n+1} &= \{ y_j \in Y_{j-1,1} \mid \exists y_{j-1} \in Y_{j-1,n+1} : y_j \prec y_{j-1} \} \text{ and} \\ Y_{j,i} &= \text{basis}(\left\{ z \in S \mid z \overset{t_i}{\hookrightarrow} Y_{j,i+1} \right\}) \end{aligned}$$

for each $n \geq i \geq 1$ and $j \geq 2$ until for some $j = k$, set $Y_{j,n+1}$ will be empty. This termination is guaranteed by Lemma 2.2 on p. 18 (well-foundedness) as we construct a strictly decreasing sequence of states $y \succ y_1 \, (\in Y_{1,n+1}) \succ y_2 \, (\in Y_{2,n+1}) \succ \cdots$ which is bound to be finite.

Assume, the $k+1$-th set $Y_{k+1,n+1}$ is empty, then the set of accelerated

[3] In our reference implementation of the algorithmic framework, the pred-basis computation is enforced to be done w.r.t. a given transition label.

predecessors we compute this way is

$$A_x = \bigcup_{j=1}^{k} Y_{j,n+1} \, .$$

5.1.3. Computation of Maximal Extensions for Petri Nets

In the setting of P/T Petri nets the construction of the maximal extensions from an acceleration candidate and a candidate sequence is cheap as the basis of covering predecessors of a marking is a singleton. Let us dip into the details needed to compute the pred-basis of Petri net markings.

Definition 5.2 (Monus [Ame84]). For natural numbers $a, b \in \mathbb{N}$, we define the *monus* operation to be

$$a \ominus b := \max \{ 0, a - b \} \, . \qquad \Diamond$$

With this definition, we can easily compute a minimal basis of covering predecessors w.r.t. a given transition.

Lemma 5.2 (PN Pred-Basis Computation). Given a PN (P, T, W), a transition $t \in T$, and a marking $m \in \mathbb{N}^P$, a minimal basis of covering predecessors of m w.r.t. transition t consists solely of the marking $m_c = (m \ominus W(t, -)) + W(-, t)$.

Proof. To show that $\{ m_c \}$ is a minimal basis as promised, we prove 1. that its successors w.r.t. t do cover m and 2. that there is no strictly smaller marking with this property.

1. The successor of m_c w.r.t. t is

$$\begin{aligned} & m_c - W(-, t) + W(t, -) \\ = & ((m \ominus W(t, -)) + W(-, t)) - W(-, t) + W(t, -) \\ = & (m \ominus W(t, -)) + W(t, -) \, , \end{aligned}$$

i.e. for each $p \in P$, the marking contains $(m(p) \ominus W(t,p)) + W(t,p)$ tokens on the place. More precisely, by Def. 5.2, for each $p \in P$, we have $\max \{ 0, m(p) - W(t,p) \} + W(t,p)$ which we transform to

$$\begin{aligned} & \max \{ 0 + W(t,p), m(p) - W(t,p) + W(t,p) \} \\ = & \max \{ W(t,p), m(p) \} . \end{aligned}$$

This means that the successor marking is greater or equal to m and therefore the marking under consideration is a covering predecessor.

2. Assume there is a marking m_s that is strictly smaller than m_c and its successor w.r.t. t covers m, formally, $m_c > m_s \overset{t}{\hookrightarrow} m$. Without loss of generality, let $p \in P$ be a place with $m_s(p) < m_c(p)$. Again w.l.o.g, we choose $m_s(p) = m_c(p) - 1$. When firing transition t from m_s we end up with

$$\begin{aligned} & m_c(p) - 1 - W(p,t) + W(t,p) \\ = & ((m(p) \ominus W(t,p)) + W(p,t)) - 1 - W(p,t) + W(t,p) \\ = & \max \{ 0, m(p) - W(t,p) \} + W(t,p) - 1 \\ = & \max \{ W(t,p) - 1, m(p) - 1 \} \end{aligned}$$

tokens on the successor marking. As this successor covers m it follows that $\max \{ W(t,p) - 1, m(p) - 1 \} \geq m(p)$ holds and therefore

$$\max \{ W(t,p) - 1, m(p) - 1 \} = W(t,p) - 1 .$$

This implies that $W(t,p)$ is strictly larger than $m(p)$. Since m_c is defined to have $(m(p) \ominus W(t,p)) + W(p,t)$ tokens on place p and with the definition of monus we have $m_c(p) = W(p,t)$. But now the successor $m_c + W(-,t) + W(t,-)$ contains exactly $m_c(p) - W(p,t) + W(t,p) = W(t,p)$ tokens on place p and we know from the proof of m_c being a *covering predecessor* of m, that its successor covers m, i.e. that $W(t,p)$ is greater than or equal to $m(p)$! Thus, $W(t,p) > m(p) \leq W(t,p)$ holds. This is a contradiction and therefore the assumption that there exists a covering predecessor w.r.t. t that is strictly smaller than m_c is false.

The minimal basis of covering predecessors of marking m consists of marking m_c only. □

With this pred-basis computation, the method to find the maximal extension of a candidate sequence σ for some marking m is simply to

1. start with $m_1 = m$ and $i = 1$ and
2. find the covering predecessor m_{i+1} of m_i w.r.t. $\sigma = t_1 t_2 \cdots t_n$ by computing
$$m_{i+1} = (\cdots((m_i \ominus W(t_n, -)) + W(-, t_n)) \ominus \cdots \\ \ominus W(t_1, -)) + W(-, t_1)\,,$$
3. if $m_{i+1} < m_i$, then m_{i+1} is an accelerated predecessor of m and we can increment i and repeat step 2,
4. else there are no more accelerated predecessors.

Summing up, in the Petri net setting, where there is no branching in the pred-basis computation, finding the maximal extension of a candidate sequence can be achieved rather conveniently.

5.2. Pruning

During backward reachability analysis, states may be explored which are backward reachable from $\uparrow F$, but are not forward reachable from $\downarrow I$, i.e, they lie in $\mathit{pre}^*(\uparrow F) \setminus \mathit{suc}^*(\downarrow I)$. Here, $\mathit{suc}(X)$ is the set of one-step successors of X and $\mathit{pre}^*, \mathit{suc}^*$ are the reflexive transitive closures of $\mathit{pre}, \mathit{suc}$. Such states do not contribute to coverability and should be avoided during the search. This is a typical problem of backward analysis methods (cf. for example [HKQ03]). To our knowledge, the first to attack this problem in the context of Petri nets and the basic BR by employing structural properties of the model under consideration were Delzanno et al. [DRV01].

For pruning, it is assumed that there is a known over-approximation $P \supseteq \mathit{suc}^*(\downarrow I)$ which is then used to cut off states from outside P during the analysis. Figure 5.6 shows the effect of pruning with an over-approximation guaranteeing that states reached backward from a, b, c, d via transition 5 are not reachable from $\downarrow I$.

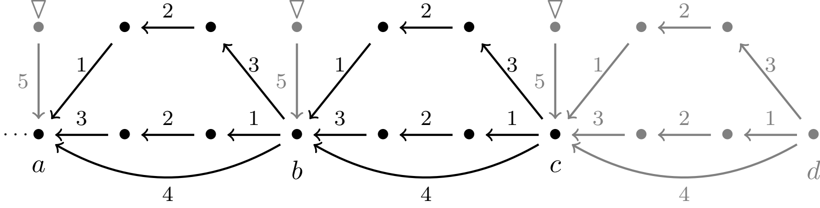


Figure 5.6.: Schematic example search space for pruning.

Definition 5.3 (Pruning). Given a WSLTS $(S, L, \rightarrow, \preceq)$, a finite set of initial states $I \subseteq S$, an over-approximation of the reachable states $P \supseteq suc^*(\downarrow I)$, and some state x, we define the *pruning* SSC by

$$opb_{prune}(x) := \downarrow P \cap pb(x)\,.$$

As only one-step predecessors are returned by opb_{prune}, function wit_{prune} is the trivial

$$wit_{prune,x}(y) := lbl_x(y)\,. \qquad \Diamond$$

Example 5.2 (Pruning for N_{ex}). In Fig. 5.7 we have drawn the search space of N_{ex} w.r.t. m_Ω where we used pruning via the structural heuristics of place invariants to reduce the number of states visited. States that are visited in the full search space (cf. Fig. 2.7 on p. 32) but not in the reduced space in this example are coloured gray.

A place invariant of the example Petri net N_{ex} w.r.t. initial marking m_α is $p_2 + p_3 = 1$, ensuring that every marking that is reachable from m_α contains exactly one token on either place p_2 or place p_3. We represent the downward-closure of this invariant by $p_2 + p_3 \leq 1$ which means that any marking m coverable from m_α satisfies the equation $m(p_2) + m(p_3) \leq 1$ and vice versa, for any marking m' with $m'(p_2) + m'(p_3) > 1$ that is explored (backwards) by our algorithm, we immediately know that is it *not coverable* from m_α.

In the search space depicted in Fig. 5.7, we see that pruning allows us to identify the states $2p_2 + p7$ and $2p_2 + p_5$ to violate the constraint $p_2 + p_3 \leq 1$, as already p_2 contains two tokens. Therefore, these states are not explored.

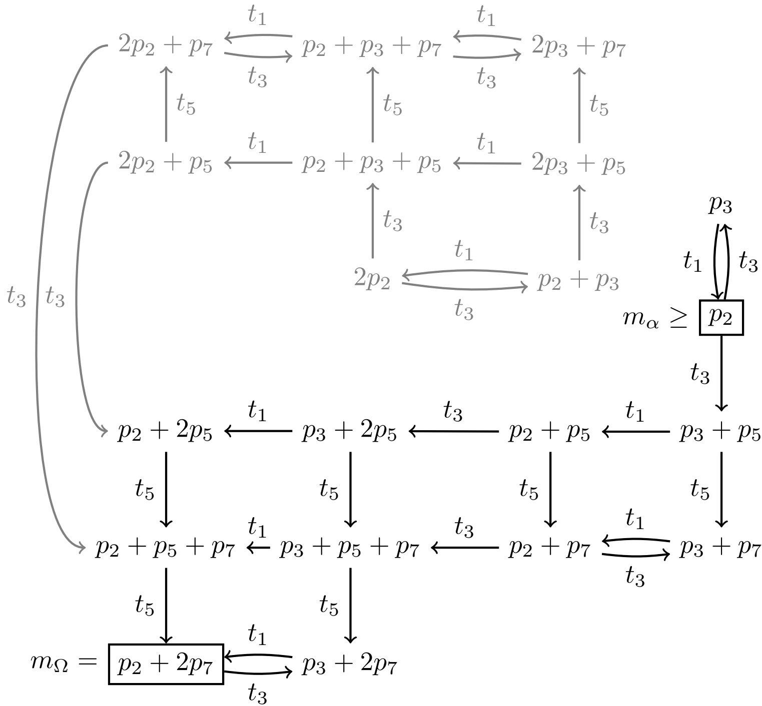


Figure 5.7.: Exploration of N_{ex}, m_Ω with pruning.

As the invariant provides only an over-approximation of the state space, there are cases where the marking is not reachable but not pruned. Consider (a new) final state $p_1 + p_6$ for example which is not coverable as both p_1 and p_6 cannot be marked at the same time. The token from p_1 has to be consumed by transition t_2 in order to produce a token on p_4 which then enables transition t_4 (if p_5 is has tokens)—the only transition that produces tokens in p_6 (cf. Fig. 2.2 on p. 14). As p_6 is unbounded, meaning that an arbitrary number of tokens can be produced on this place, Petri net theory (cf. for example [STC98]) tells us that there is

no P-invariant with a non-zero weight for p_6.[4] The result is that pruning via P-invariants cannot reject the marking $p_1 + p_6$. ◊

Lemma 5.3. Pruning is a search space construction.

Proof. Let $(S, L, \rightarrow, \preceq)$ a WSLTS, $I \subseteq S$ a finite set of initial states, $x \in S$ a state, and $P \supseteq suc^*(I)$ an over-approximation of the reachable state space.

- Since $opb_{prune}(x) \subseteq pb(x)$, it is finite, i.e. condition **Fin** is satisfied.
- For case $dist(\downarrow I, \uparrow x) = \infty$ condition **Dist** is trivially satisfied.

  Let $dist(\downarrow I, \uparrow x) = k \in \mathbb{N}$. The definition of *dist* guarantees the existence of a $y \in pb(\uparrow x) = \uparrow pb(x)$ with $dist(\downarrow I, y) = k - 1$. In order to show satisfaction of **Dist**, we prove that y does not get pruned, i.e. $y \in \uparrow opb_{prune}(x) = \uparrow(\downarrow P \cup pb(x)) = \uparrow(\downarrow P \cup \uparrow pb(x))$: For y to lie in the intersection, $y \in \downarrow P$ has to hold as $y \in \uparrow pb(x)$ already applies.

  From $P \supseteq suc^*(I) \supseteq I$ and $\downarrow P \supseteq P$ follows $\downarrow P \supseteq \downarrow I$. Together with $dist(\downarrow I, y) = k - 1$, which implies that y is reachable from $\downarrow I$, it follows that $y \in \downarrow P$. Thus $y \in \downarrow P \cap \uparrow pb(x)$ and moreover $y \in \uparrow opb_{prune}(x)$.

  Therefore, condition **Dist** is satisfied by opb_{prune}.
- The $wit_{prune,x}$ function is simply lbl_x, for which we know that it satisfies **Wit**.

We conclude that pruning is a search space construction. □

We stress that pruning depends on an over-approximation of the concrete model under consideration, i.e, the technique requires knowledge of the system class. As mentioned in Sect. 5.1 on backward acceleration, our experiments (cf. Sect. 7.2 on p. 207) show that knowledge gained by static analysis of the model under consideration, like pruning by structural heuristics as P-invariants, has a rather high positive impact on the running time of our reference implementation.

[4] More precisely, place p_6 is not *structurally bounded*. If a place is structurally bounded, it is bounded for any initial marking. Of course, p_6 is bounded if the initial marking is empty as N_{ex} cannot fire any transition in that case.

As shown in Example 5.2 for Petri nets—and even Petri nets with transfer—, over-approximations of the search space can be obtained by classical P-invariants [STC98, Cia94].[5] For LCSs, one can use symbolic configurations that represent alternative channel contents by simple regular expressions [ABJ98]. This data structure is then wrapped into an EEC [GRV06b] style over-approximation.

5.3. Partial-Order Reduction

Partial-order reduction (POR) tries to avoid the exploration of several interleavings of *independent* transitions [Val90, Pel93, Pel96, ABH+97]. In the context of backward reachability analysis, this means we perform a selection of direct predecessors rather than inspecting them all [AJKP98].

Figure 5.8 shows that from the three possible states leading to a, only one is explored: the transition sequences 4, 1 2 3, and 3 2 1 were identified to lead to the same state and two of them were discarded.

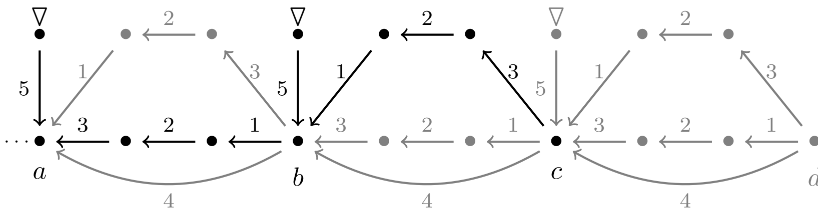


Figure 5.8.: Schematic example search space for POR.

Definition 5.4 (Partial-Order Reduction). Consider $(S, L, \rightarrow, \preceq)$ a WSLTS, a finite set of initial states $I \subseteq S$, and a state $x \in S$. *Partial-order reduction* relies on a function *choose* : $S \longrightarrow \mathcal{P}(L)$ which maps

[5]The computation of good, i.e. "minimal semi-positive" P-invariants efficiently is an important subject of research—especially with their connection to so-called *siphon-trap* structures in Petri nets. See for example [CS91, TITW05, OWW10, DT88].

a state to a set of transition labels. The function has to satisfy the following constraints in order to yield a correct algorithm [AJKP98][6]:

1. Taking a transition with a chosen label backwards from state x yields a smaller state than taking a transition with a label not chosen.

   In detail, every selected transition label $\ell \in \mathit{choose}(x)$ has to be independent from each sequence $\sigma \in (L \setminus \mathit{choose}(x))^*$ of labels not selected. Independence means that for any partition $\tau_1\tau_2$ of σ, state y reached backwards from x with $\tau_1\tau_2\ell$ has to be smaller than or equal to state y' reached backward from x with ℓ postponed, i.e. with sequence $\tau_1\ell\tau_2$. More formally, with the above x, y, y', and ℓ,

$$\forall \tau_1, \tau_2 \in (L \setminus \mathit{choose}(x))^* : x \xleftarrow{\tau_1\tau_2\ell} y \leq y' \xhookrightarrow{\tau_1\ell\tau_2} x\,.$$

2. There is no sequence $\sigma \in (L \setminus \mathit{choose}(x))^*$ of transition labels that are not chosen but with which $\uparrow x$ is reachable from $\downarrow I$.

   Technically, for any $\sigma \in (L \setminus \mathit{choose}(x))^*$ and any set $X \subseteq S$, $\downarrow I \overset{\sigma}{\hookrightarrow} X$ implies $X \cap \uparrow x = \emptyset$.

Partial-order reduction then keeps those states of $\mathit{pb}(x)$ that result from backward application of transitions with selected labels:

$$\mathit{opb}_{\mathit{por}}(x) \stackrel{df}{=} \{\, y \in \mathit{pb}(x) \mid \mathit{lbl}_x(y) \cap \mathit{choose}(x) \neq \emptyset \,\}\ .$$

As only one-step predecessors are computed, the $\mathit{wit}_{\mathit{por}}$ function is the trivial

$$\mathit{wit}_{\mathit{por},x}(y) := \mathit{lbl}_x(y)\,. \qquad \Diamond$$

Example 5.3 (Partial-Order Reduction for N_{ex}). In Fig. 5.9 we have drawn the search space of N_{ex} w.r.t. m_Ω where we used POR to reduce the number of states visited. States that are visited in the full search space (cf. Fig. 2.7 on p. 32) but not in the reduced space in this

[6] In [AJKP98] it is stated that the second condition on *choose* can be replaced by three other conditions leading to a more general version. However, for sake of brevity, we stick to the simpler, more specialized version. Please see [Pel98] for an overview of POR methods.

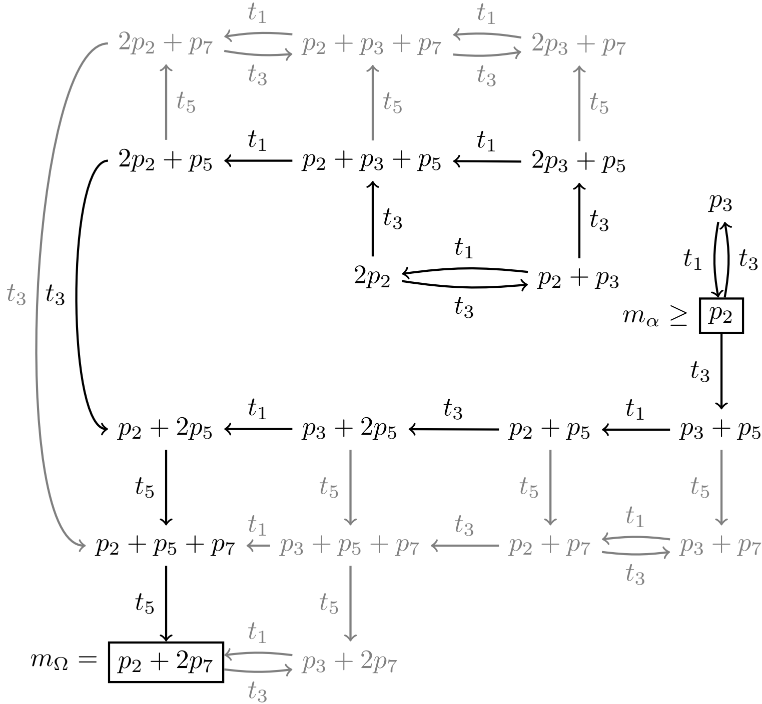


Figure 5.9.: Exploration of N_{ex}, m_Ω with partial-order reduction.

example are coloured gray. We assume it is always allowed to postpone transitions t_1 and t_3 in relation to t_5, i.e. if minimal covering predecessors are reached backwards via t_1 or t_3 and t_5, then only the predecessor w.r.t. t_5 remains in the optimized predecessor basis. For example, from the final marking $p_2 + 2p_7$, there are two markings, $p_2 + p_5 + p_7$ (via t_5) and $p_3 + 2p_7$ (via t_1), in the minimal basis of covering predecessors. As transition t_1 can be postponed w.r.t. t_5, the *choose* operation returns $\{ t_5 \}$ and only covering predecessor $p_2 + p_5 + p_7$ is retained in the basis of optimized predecessor. Observe that every path from the final state to the initial state contains the chosen transition label t_5 so that the second constraint for partial-order reduction is satisfied.

The effect in this specific Petri net example is rather drastic: except for one detour from $p_2 + 2p_5$ backwards via t_3 into a part of the search space which is not reachable from the initial marking, the explored search space is almost reduced to only the sequence of transition labels with which the final state is coverable (marking p_3 is explored, too).
As an example for an invalid *choose* operation, consider that t_5 is chosen over t_3 and t_3 is chosen over t_1. In this case, the path from $p_2 + p_5 + p_7$ to the initial marking would be not be explored in the search, as only transition t_3 would be taken backwards to marking $2p_2 + p_5$. This would violate constraint 2 of Def. 5.4. $\diamondsuit$

Lemma 5.4. Partial-order reduction is a search space construction.

Proof. Let $(S, L, \rightarrow, \preceq)$ a WSLTS, $x \in S$ a state, and $choose(x)$ a selection function for partial-order reduction.

- Since $opb_{por}(x) \subseteq pb(x)$, it is finite, i.e. condition **Fin** is satisfied.
- For case $dist(\downarrow I, \uparrow x) = \infty$ condition **Dist** is trivially satisfied.

  Let $dist(\downarrow I, \uparrow x) = k \in \mathbb{N}$. Let $\sigma = \ell_1 \cdot \cdots \cdot \ell_k \in L^*$ be a sequence from $y' \in \downarrow I$ to $\uparrow x$, i.e. $y' \overset{\sigma}{\hookrightarrow} \uparrow x$. By property 2 there is an $\ell_i \in choose(x)$ with i minimal. By property 1, the state y reached backward from $\uparrow x$ by postponing ℓ_i is less equal to y', i.e. $y \xhookrightarrow{\ell_1 \cdot \cdots \cdot \ell_{i-1} \cdot \ell_{i+1} \cdot \cdots \cdot \ell_k \cdot \ell_i} x \Rightarrow y \leq y'$, and thus $y \in \downarrow I$.

  As ℓ_i is in $choose(x)$, no labels of transitions on a path to $\downarrow I$ are neglected by $opb_{por}(x)$. Furthermore, as $opb_{por}(x) \subseteq pb(x)$, the distance to $\downarrow I$ decreases: $dist(\downarrow I, opb_{por}(x)) < k$.
- The $wit_{por,x}$ function is simply lbl_x, for which we know that it satisfies **Wit**.

We conclude that partial-order reduction is a search space construction. $\square$

As for pruning, we stress that partial-order reduction is an optimization that takes into account specific knowledge about the system class or about the concrete model under consideration. For Petri nets, the dependence of transitions depends on the net's structure as well as the current marking for which its covering predecessors have to be computed.

The technical details are presented and discussed in [AJKP98]. An instantiation for lossy channel systems (LCSs) is introduced in [AKP97] by three simple rules: 1. Operations of the same process are dependent. 2. A receive operation is chosen over a send operation if they use the same channel, the same message and the channel is empty. 3. All other operations are independent s.t. *choose* may make an arbitrary pick. To understand the second rule, we turn to the computation of a minimal basis of covering predecessors as stated in [AKP97, Def. 4].

Definition 5.5 (LCS Pred-Basis Computation). Let $(Q, C, M, \rightarrow)$ be a lossy channel system and (q, W) be a configuration of that LCS with control states $q_1, q_2 \in Q$ and channel content $W_2 \in {M^*}^C$. Furthermore, let $c \in C$ be a channel and $m \in M$ be a message. The only configuration in the minimal pred-basis of (q_2, W) depends on the considered transition's operation.

- In case the transition is a local operation, i.e. $q_1 \xrightarrow{\tau} q_2$, then the covering predecessor configuration is (q_1, W).
- If the transition is a receive operation, i.e. $q_1 \xrightarrow{c?m} q_2$, then the covering predecessor configuration is $(q_1, W[c := m \cdot W(c)])$.
- For a send operation, i.e. $q_1 \xrightarrow{c!m} q_2$, then the covering predecessor configuration is one of two possible: 1. If $W = v \cdot m$ for some $v \in M^*$, then the predecessor is $(q_1, W[c := v])$, 2. otherwise (q_1, W) is obtained which means that message m has been lost after sending. $\diamondsuit$

From this definition, we see why the second rule for partial-order reduction in LCSs favors receive operations over send operations in the specific case that the message introduced (backwards) by the receive operation can then be consumed by the send operation. Here, the POR's core idea is to omit the backward execution of a send operation on an empty channel (which does not change the channel content) followed by a receive operation that produces the same message on the channel. In the execution order enforced by POR, the message is first produced and then consumed, leading to a smaller configuration.

5.4. Combination of Search Space Constructions

In order to build a fast decider for the coverability problem, we are interested in exploiting more than one SSC. Care has to be taken when combining SSCs in order to construct a stronger SSC: While the intersection of pruning and POR, $(opb_{prune} \cap opb_{por}, lbl)$, is an SSC because both optimized predecessor functions give subsets of pb and they are compatible, the intersection of pruning, POR, and the backward acceleration, $(opb_{prune} \cap opb_{por} \cap opb_{accel}, wit_{accel})$, is not. The result of backward acceleration may lie outside of pb, so the intersection can be empty.

In our tool, we found that chaining yields best results when the three SSCs are combined s.t.

(1) first, partial-order reduction is applied, leaving $opb_{por}(x)$, then

(2) a variation of pruning is applied on $opb_{por}(x)$, leaving

$$opb_{prune'}(x) = {\downarrow} P \cap opb_{por}(x)\,,$$

and finally,

(3) backward acceleration is modified to be applied on $opb_{prune'}(x)$, in the sense that $pb(x)$ is replaced by $opb_{prune'}(x)$.

Thus, a partial-order reduction is performed on the predecessors of some state, leaving few; of these predecessors, those contradicting an over-approximation get pruned, and the remaining states are accelerated. This chaining is particularly useful for an implementation of backward acceleration if the POR is deterministic in choosing certain transitions over others, leading to more homogeneous transition sequences between states, making them easier to exploit.

While it is sufficient to compute the predecessor basis of x w.r.t. POR and then remove states according to the pruning SSC from an implementation-focused perspective, to show that the proposed combination of partial-order reduction, pruning and backward acceleration is an SSC, we use that the result is the same when backward acceleration is applied on the intersection $opb_{prune}(x) \cap opb_{por}(x)$.

Definition 5.6 (Combined Optimization PORPA). Consider some state x of a WSLTS $(S, L, \rightarrow, \preceq)$. Let Z be a shorthand for $opb_{por}(x) \cap opb_{prune}(x)$. Let A_x be a subset

$$A_x \subseteq \{\, y' \in pre^*(\uparrow Z) \mid \exists y \in Z, x' \in S, \sigma \in L^*, k \in \mathbb{N} : \\ y' \prec y \prec x' \wedge y' \overset{\sigma^k}{\hookrightarrow} y \overset{\sigma}{\hookrightarrow} x' \,\}\,.$$

The optimized predecessor function PORPA is defined as

$$opb_{porpa}(x) := minimize(A_x \cup Z)\,.$$

The corresponding witness function maps each $y' \in opb_{porpa}(x)$ to either $lbl_x(y') = wit_{por,x}(y') = wit_{prune,x}(y')$ if y' is a one-step predecessor of x or to the path $\sigma^k \cdot lbl_x(y)$, where σ and y are used in the computation of set A_x. ◊

Lemma 5.5. The combined optimization PORPA is an SSC.

Proof. The proof is analogous to the one from Proposition 5.1 via the argument that $X = opb_{por}(x) \cap opb_{prune}(x)$ is distance reducing as POR is an SSC and pruning only removes predecessors that are unreachable (cf. Lemma 5.4 and Lemma 5.3). □

Example 5.4 (Effect of Combined SSC PORPA for N_{ex}). The search space of N_{ex} w.r.t. m_Ω where we used the combined optimization PORPA to reduce the number of states visited is depicted in Fig. 5.9. States that are visited in the full search space (cf. Fig. 2.7 on p. 32) but not in the reduced space in this example are coloured gray.
For the partial-order reduction part of PORPA (cf. Fig. 5.9) we assume it is always allowed to postpone transitions t_1 and t_3 in relation to t_5, resulting in following transition t_5 backwards from from m_Ω to $p_2+p_5+p_7$ and further to $p_2 + 2p_5$ before any other marking is explored.
From the pruning component of PORPA (cf. Fig. 5.7) we know that $2p_2 + p_5$—which would be reached from $p_2 + 2p_5$ via t_3—does not lie in the reachable states space's over-approximation via the downward-closure of the net's structural P-invariant $p_2 + p_3 = 1$ as the sum of the tokens on places p_2 and p_3 is larger than 1.
The remaining predecessor $p_3 + 2p_5$ is examined. It has only one predecessor in its minimal pred-basis: p_2+p_5. This marking is not added to the

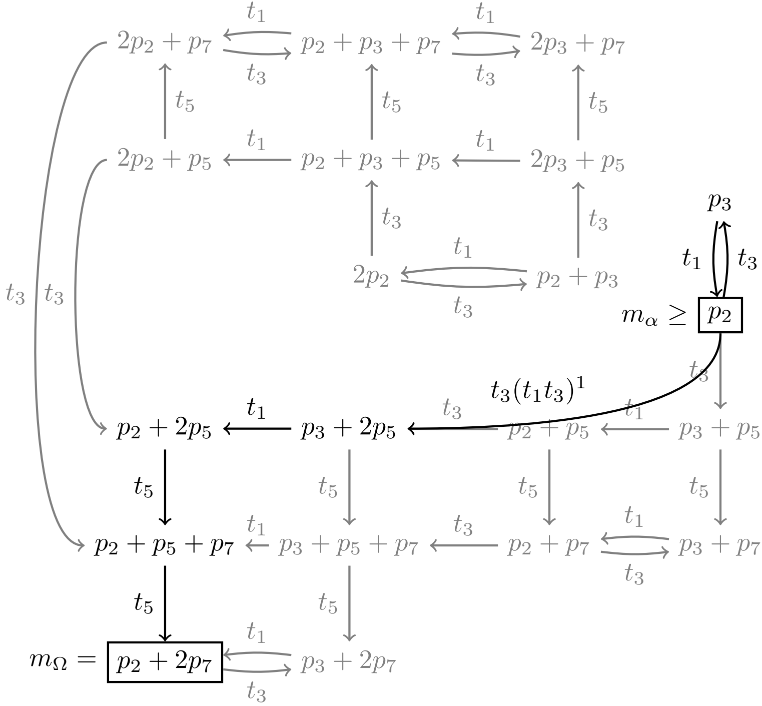


Figure 5.10.: Exploration of N_{ex}, m_Ω with combined SSC PORPA.

search space as the backward acceleration part of PORPA (cf. Fig. 5.4) kicks in and jumps directly to marking p_2 via sequence $t_3\,(t_1\,t_3)^1$.

In conclusion, the search space explored with PORPA is much smaller than with any of the optimizations alone—at least in this example. Also we can observe that the effect of the model-independent backward acceleration appears to be less pronounced (two states omitted vs. twelve states omitted) than the effects of the two optimizations that base on model-specific knowledge. $\Diamond$

5.5. Guided Search

As we have seen in the previous section, methods that reduce the backward search space while retaining paths (or at least one path) to the initial states if they exist, are highly important for the performance of the search algorithm. In fact, as we will see in the experiments of Sect. 7.2 on p. 207, a blind search without any optimizations does not suffice to answer even "simple" coverability queries.

While the optimizations of the previous section actively decrease the number of explored states, there is another well-known technique to increase the analysis' performance: In *guided search* the analysis is directed via a *search strategy*[7] to explore certain states before others, hoping that the initial states are reached earlier.

In our algorithmic framework (cf. Alg. 3.7 on p. 80) we allow for search strategies to be implemented in the selection function *select* which chooses a state from a basis of an upward-closed set. The only condition it has to satisfy to be adequately instantiated is $select(W) \in W$—where W is a finite basis of the states that are to be explored next—and thus allows for an arbitrary order for the states returned from W.

The order in which elements are chosen from W does neither influence correctness nor termination of the algorithm. However, certain orderings are preferable as they lead to fewer loop iterations and in the best case—if $\uparrow F$ is reachable from $\downarrow I$—a corresponding transition sequence is explored backwards immediately.

Let $(S, L, \rightarrow, \preceq)$ be a WSLTS, $I \subseteq S$ a finite set of initial states, $F \subseteq S$ a finite set of final states, $x, y \in S$ some states, and $\gamma_x, \gamma_y \in L^*$ sequences of transition labels with $x \xhookrightarrow{\gamma_x} F$ and $y \xhookrightarrow{\gamma_y} F$. In the context of our algorithmic framework, we have the function T that gives sequences of transition labels from elements of W to the final states and thus we may have $\gamma_x = T(x)$ and $\gamma_y = T(y)$. Several common search strategies are imaginable under the assumption that *select* picks an element minimal w.r.t. a partial order $\leq$:

DFS) Let $x \leq y :\Leftrightarrow |\gamma_x| \geq |\gamma_y|$. The selection function chooses a state from W where the trace length is maximal, hence leading to a *depth-first search*.

[7] See [LCL87] for some overview of early strategies.

BFS) Let $x \leq y :\Leftrightarrow |\gamma_x| \leq |\gamma_y|$. The selection function chooses a state from W where the trace length is minimal, hence leading to a *breadth-first search*.

BF) If we take a heuristic function $h(x)$ that estimates (but does not overestimate) the number necessary transition steps from $\downarrow I$ to the state under consideration, we end up with a so-called greedy *best-first search* and have $x \leq y :\Leftrightarrow h(x) \leq h(y)$.

A^*) Consider a heuristic function $h(x)$ that is *consistent*, i.e, it estimates (but does not overestimate) the minimal trace length from $\downarrow I$ to state x and satisfies the triangle inequality. In other words: for any predecessor $y \hookrightarrow x$ of x, value $h(x)$ does not exceed $dist(y, \uparrow x) + h(y)$ and every state in $\downarrow I$ is mapped to 0 [HNR68]. Let $x \leq y :\Leftrightarrow |\gamma_x| + h(x) \leq |\gamma_y| + h(y)$. The search order followed by the algorithm then mimics the well-known A^* search [HNR68] which combines the approaches of greedy best-first search and breadth-first search.

In the context of BF and A^*, there exists a vast number of approaches for such heuristic distance estimates. To name a few, the authors of [DFP06, KHDB06, KHL08] propose to infer distance from certain abstractions (on-the-fly), in [YD98]—among other ideas—the Hamming distance between bit-vector representations of states is employed, and in the more current approach of [WKP09], Wehrle et al. turn the focus on so-called "useless transitions" which allow to provide a finer-grained distance heuristics.
For our example classes of WSLTSs—Petri nets (with transfer) and lossy channel systems (LCSs)—we can formulate a lower bound on the distance of a state to $\downarrow I$ based on the model's syntax.

5.5.1. Syntactic Distance and Syntactic Weight

For Petri nets (with transfer), the *syntactic distance* of a marking m to the downward-closure of initial states is determined by the maximal syntactic distance of the places marked in m. Here syntactic distance of a place to $\downarrow I$ is the number of transitions in the shortest path from that place to the set of transitions with empty presets and to places that are marked initially. Transitions with empty presets have to be included as targets for the heuristic because they provide short-cuts to smaller

markings in the context of backward analysis. The syntactic distance is lifted to markings by taking the maximum syntactic distance of the places with positive token count in that marking.

As an example, consider a PN and a marking m where the shortest path from m to the initially marked places has at least two transitions. Assume further that there is a transition t with $W(t, -) = m$ and an empty preset. Clearly, the empty marking m_e is reached as a predecessor of m under t and $m_e \in \downarrow I$, taking only one step.

We choose to enhance the heuristic by taking into account the effect of transitions, upgrading $\leq$ to a lexicographical ordering: First, states are compared w.r.t. the syntactic distance and if the distances are equal, we aim to select that state where the transitions seem to consume more tokens when fired backward. Therefore, we infer the number of tokens each transition produces by subtracting the number of incoming arcs from the number of outgoing arcs which we call the transition's *syntactic weight*. The syntactic weight of a marking is the sum of the number of tokens on each place multiplied by the syntactic weights of the transition in the preset of that place. While the notion of syntactic weight is rather coarse, it tends to improve our search.

Example 5.5 (PN Syntactic Distance and Weight). Figure 5.11 depicts our running example N_{ex} where places have been annotated with their syntactic distance and transitions have been annotated with syntactic distance and syntactic weight. In the postset of initially marked places p_1 and p_2 are transitions t_2 and t_3. Therefore, these transitions have syntactic distance 1. As t_2 has two incoming arcs and one outgoing arc, the transition consumes one token more than it produces and its syntactic weight is -1. Transition t_3 on the other hand produces one token more than it consumes and has syntactic weight 1. Consider markings $p_2 + 2p_5$ and $p_3 + 2p_5$. The maximal syntactic distance is $d = 1$ in both cases, so the syntactic weight of the transitions in the preset of the marked places is observed. For marking $p_2 + 2p_5$ the transitions are t_1 and t_3 with a total weight of $w = 2$ (two times the weight of t_3). For the other marking, transition t_3 is in the preset of p_3 and of p_5 and therefore the syntactic weight is $w = 3$ as there are three tokens in the postset of t_3. Hence, we roughly assume that t_3 may consume more tokens when fired backward and select marking $p_3 + 2p_5$ over $p_2 + 2p_5$. Indeed, this marking is closer to the initial marking $p_1 + p_2$. $\diamondsuit$

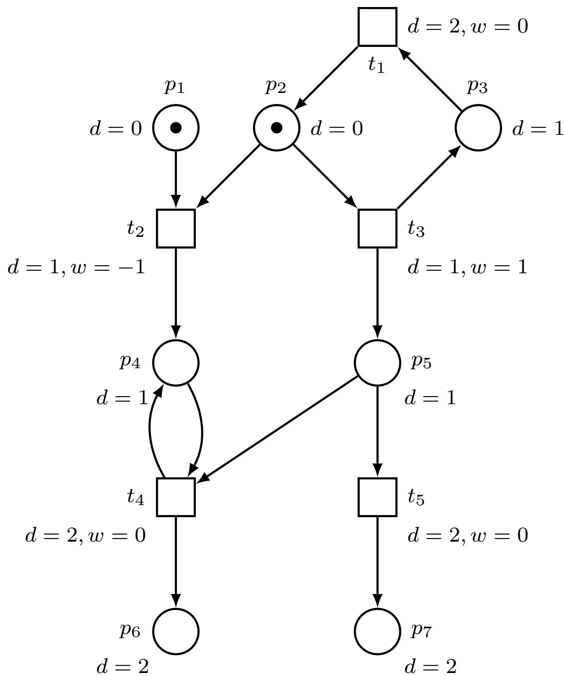


Figure 5.11.: Syntactic distance and weight in Petri net N_{ex}.

In the LCS case, the syntactic distance of a configuration (q, γ) to the initial states is the length of the shortest sequence of transitions leading from q_0 to q, thus ignoring channel contents. This means that the distance heuristics and identifies the minimal number of transitions necessary to reach the initial control state of each automaton in the LCS. From this set of minimal distances the maximum is taken, representing a minimal number of transitions necessary to reach all initial control states of the LCS. The syntactic distance for LCS is a close relative of the “local state distance” of Edelkamp et al. who consider concurrent finite state machines [ELLL01, ELLL04].
Our experiments in Sect. 7.2 on p. 207 show an altogether improved performance when using BF with the syntactic distance and local state distance heuristics as compared to breadth-first search.

CHAPTER SIX

Data Structures

When you make something, cleaning it out of structural debris is one of the most vital things you do.
— Christopher Alexander, Architect

Contents

6.1 Related Work 160
6.2 Operations 161
6.3 From Necessary Conditions to Equivalence Classes . 164
6.4 Equality Conditions 167
6.5 Total Order Conditions 168
6.6 Subset Conditions 170
6.6.1 Full Powerset Search Trees 172
6.6.2 Relaxed Powerset Search Trees 176
6.6.3 Preliminary Experiments 184
6.7 Hierarchy of Necessary Conditions 185
6.7.1 Preliminary Experiments 187
6.8 Select Operation 189

In our algorithmic framework (Sect. 3.3 on p. 79), we modify upward-closed sets (UCSs) via finite bases. For reasons of efficiency we intend to have minimal (and thus finite, cf. Lemma 2.6 on p. 22) bases where the elements form an anti-chain, i.e. they are incomparable w.r.t. the underlying well-quasi ordering (WQO). While an implementation of our framework is free to choose an arbitrary data structure for minimal bases of upward-closed sets, we propose to employ a simple and generic data structure that can be readily adapted to the different concrete models. We begin by inspecting some well-known data structures used in the context coverability analysis. While these are all specialized for certain concrete models of well-structured transition systems (WSTSs), we strive for integrated techniques applicable to several models.

6.1. Related Work

Some earlier approaches to store large sets of states (with a main focus on Petri net markings and reachability problems) come from Ciardo and Miner [CM97] who partitioned the model under consideration and used multiple levels of search trees in the style of Chiola [Chi90], one for each partition. This led to a decrease in running time in comparison to the use of a single search tree. Building on their results, Miner and Ciardo [MC99] then employed generalizations of *binary decision diagrams* (BDDs) [Bry86] to arbitrary integer functions on integer variables, so-called *multi-valued decision diagrams* (MDDs, cf. [SKMB90]), and structured the model under consideration into partitions each of which is covered by an MDD.
Different approaches for the efficient handling of sets of tuples (e.g. Petri net markings) where undertaken by Delzanno et al. who introduced *covering sharing trees* [DR00, DRV02][1] and Ganty et al. who extended the idea to *interval sharing trees* [GMV$^+$07]. Both techniques build upon the *sharing trees* of Zampunieris and Le Charlier [ZL95] that use directed acyclic graphs with a root and a terminal node to “share” common parts of tuples. In sharing trees, each path from the root node to the terminal node stands for a tuple in the set and checking if a tuple is contained in

[1] Finkel et al. later translated the idea to *downward-closed covering sharing trees* [FRSV03] to allow for forward coverability analysis. The prefix sharing of the sharing tree data structure relates to Chiola’s multi-level technique [Chi90].

the upward-closure of the set represented by the tree requires time linear in the number of edges of that tree.
The framework of Bingham and Hu [Bin05, BH05] uses BDDs as its fundamental data structure for a backward reachability analysis of a subclass of well-structured transition systems.
Most of the previously mentioned data structures aim for compression of the sets they represent and increase the program's performance via segue way of reduced data size. They are complex data structures and intricate operations are necessary to ensure that integrity and normal forms are maintained.

6.2. Operations

In our work, we intend to provide data structures that are easily employable in an implementation of our algorithmic framework, that are extensible in the sense that interfaces are lightweight, and that allow for simple proofs. Therefore, we choose to focus on structuring minimal sets (anti-chains) of states by basic rules rather than to represent them most compactly. To identify the operations needed on the underlying data structure, we recall how our algorithmic framework Alg. 3.7 on p. 80 accesses finite bases V and W:

- Set W to be a minimal basis of the final states.
  $W := \mathit{minimize}(F)$
- Test if W is non-empty.
  while $W \neq \emptyset$ **do** ...
- Select and remove a state x from W.
  $x := \mathit{select}(W);\ W := W \setminus \{\, x \,\}$
- Add a state x to the basis V and keep the basis minimal.
  $V := \mathit{minimize}(V \cup \{\, x \,\})$
- Test if a state y is not in the upward-closure of V.
  if $y \notin \uparrow V$ **then** ...
- Add a state y to the basis W and keep the basis minimal.
  $W := \mathit{minimize}(W \cup \{\, y \,\})$

- Assign V (and W) explicitly.
  $V := \{ x \}$ and $W := \emptyset$

With these program statements in mind, we choose to allow for access to the data structure we are constructing via the following basic operations where we consider some minimal basis X and a state x. To allow for an efficient implementation of the select statement $x := \mathit{select}(W)$ in the context of data structures, we aim to impress a total ordering on the states in W as proposed in Sect. 5.5 on p. 155. In Sect. 6.8 we discuss our choice of storing X both for access via the *select* function and the following operations.

1. Clear
   $X := \emptyset$

2. Test if empty
   $X \stackrel{?}{=} \emptyset$

3. Add an element
   $X := X \cup \{ x \}$

4. Remove an element
   $X := X \setminus \{ x \}$

5. Given state x, remove all elements covering x
   $X := X \setminus {\uparrow} x$

6. Given state x, test if an element covered by x is contained
   $x \stackrel{?}{\in} {\uparrow} X$

Note that $X := \mathit{minimize}(X \cup \{ x \})$ is split into to the operations of ensuring that x is not in ${\uparrow} X$, removing elements covering x by $X := X \setminus {\uparrow} x$, and lastly adding x to set X. This way, even the statement $W := \mathit{minimize}(F)$, which can be replaced by

$$W := \emptyset;\ \textbf{foreach}\ x \in F\ \textbf{do}\ W := \mathit{minimize}(W \cup \{ x \})\ \textbf{od}\,,$$

is supported and $V := \{ x \}$ can be handled analogously.

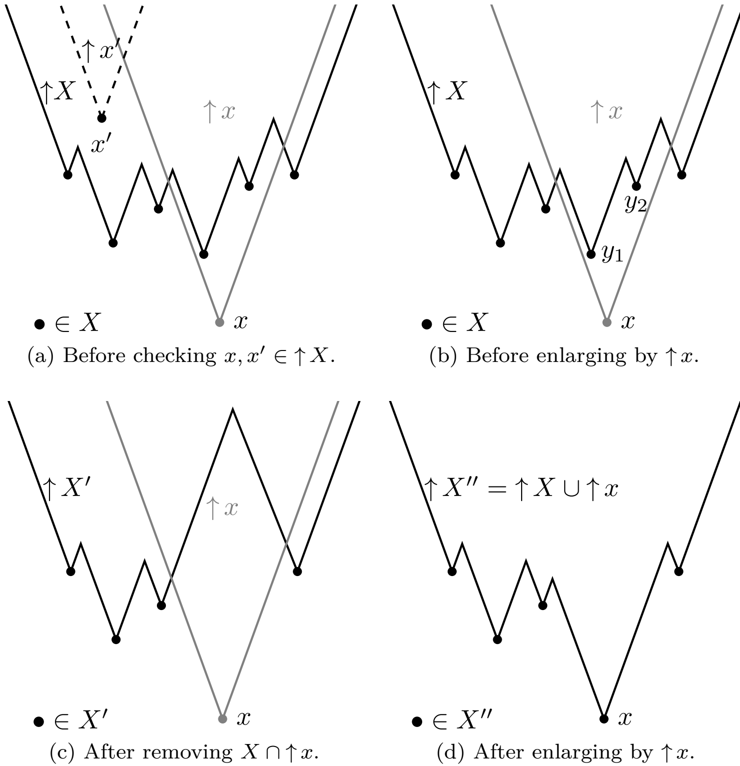


Figure 6.1.: Updating minimal basis X with x, x'.

Example 6.1. The intuition is to have a structure storing a UCS $\uparrow X$ in a way which, given a UCS $\uparrow x$, allows for fast check of $x \notin \uparrow X$, and, in case that the check succeeds, the computation of $\{ y \in X \mid y \succeq x \}$, which is $X \cap \uparrow x$ as $\uparrow x \nsubseteq \uparrow X$ holds. If $\uparrow X$ is to be enlarged by $\uparrow x$, the set $\uparrow X \cap \uparrow x$ of elements subsumed by $\uparrow x$ is removed from the minimal basis X and x is added. Figure 6.1 shows the process of enlarging $\uparrow X$ by $\uparrow x$ graphically. Black dots represent the elements of X and the area above of black lines indicates $\uparrow X$. The area above the gray lines indicates $\uparrow x$. Figure 6.1a depicts the relationship between two elements x, x' and X. While x' lies above of some element of X and thus $x' \in \uparrow X$, element x lies below all of X's elements, so that $\uparrow x \nsubseteq \uparrow X$. As $\uparrow x'$ is already a subset of $\uparrow X$, we discard it (cf. Fig. 6.1b). To build the union of $\uparrow X$ and $\uparrow x$, thus enlarging X, the two elements y_1, y_2 of X that are subsumed by x are removed from X (cf. Fig. 6.1c). The minimal basis of $\uparrow X \cup \uparrow x$ contains x and all elements of X except y_1, y_2 (cf. Fig. 6.1d). $\Diamond$

As a starting point, consider a very basic but general implementation of bases for upward-closed sets: a linked list of minimal elements in the set. To test for $x \in \uparrow X$ we iterate over each $y \in X$ and check for $y \preceq x$. Removing non-minimal elements, adding a minimal element, and the other operations are similarly simple. However, to improve performance, we are interested in structuring the set to relieve us from inspecting each of its elements.

6.3. From Necessary Conditions to Equivalence Classes

Our idea to reduce the number of elements that have to be examined when testing inclusion of a state in an upward-closed set is to employ necessary conditions for states to be in WQO relation. We want to be able to efficiently divide the UCS's minimal basis into those states that satisfy the necessary conditions and those that do not—which in turn do not have to be examined further.

Take Petri net (PN) markings for example: $m_1 \leq m_2$ implies that the number of tokens in m_1, $tok(m_1)$, does not exceed the number of tokens in m_2, $tok(m_1) \leq tok(m_2)$. This necessary condition gives rise to a simple equivalence relation on a set of markings: two markings are in the same

equivalence class if they contain the same number of tokens. To test for $x \in \uparrow X$ only those equivalence classes have to be checked where the number of tokens does not exceed the number in x. Equivalence classes of markings with more tokens are skipped as the necessary condition is violated.
As a generalization, let $\preceq$ be a WQO, x and x' some states, and

$$x \preceq x' \quad \text{imply} \quad \varphi_1(x, x') \wedge \cdots \wedge \varphi_n(x, x') .$$

We structure the necessary conditions $\varphi_1, \ldots, \varphi_n$ to form a hierarchy in the sense that a set X of states is divided into equivalence classes $X_{\varphi_1}^{(1)}, \ldots, X_{\varphi_1}^{(k_1)}$ by φ_1, and we proceed to divide each $X_{\varphi_1}^{(i)}$ into equivalence classes with respect to φ_2 and so forth. Under the assumption that accessing the equivalence class of a state x is efficient, the performance the operations identified in the previous section may be drastically improved.
Let us turn to formal definitions of what we call *indicator functions* and equivalence classes of functions.

Definition 6.1 (Indicator Function). Let $\preceq$ be a well-quasi ordering on X, $f : X \longrightarrow Y$ be a function and $x, x' \in X$. We call f an *indicator function* if there exists a relation $\sim \subseteq Y \times Y$ with $f(x) \sim f(x')$ being a necessary condition for $x \preceq x'$ and a value $f(x)$ is called *indicator value.* $\diamondsuit$

Before we will inspect partitions of finite bases w.r.t. indicator functions in the following sections, we introduce the notions of induced equivalence relations and blocks.

Definition 6.2 (Equivalence Relation Induced by a Function). Let Z, Y be arbitrary sets, $f : Z \longrightarrow Y$ be a function, and $x, y \in Z$. The *equivalence relation induced by function* f is defined as $x \sim_f y :\Leftrightarrow f(x) = f(y)$. We call $[x]_{\sim_f}$ the *equivalence class of* x *w.r.t. function* f and write $[x]_f = \{\, x' \in Z \mid f(x') = f(x) \,\}$ as a shorthand. It is the set of elements with the same image as x, or simply put $f^{-1}(f(x))$.
The induced equivalence $\sim_f$ partitions any set $X \subseteq Z$ into the *quotient set* $X/\sim_f$ consisting of the subsets of the equivalence classes of $\sim_f$ that lie in X, i.e. $[x]_f \cap X$ for $x \in X$. We call these parts of equivalence classes *blocks of* X *w.r.t.* f, or simply *blocks* if the referenced set and function are clear from the context. $\diamondsuit$

While several types of necessary conditions can be conceived of—especially with specific models in mind—, in the following we present three classes of generic conditions which utilize indicator functions like $tok(m)$:

- *Equality* conditions, which use equality of indicators of states. Take LCSs for example: The well-quasi ordering requires the control states of two configurations to match (and the channel states to be in Higman's subword ordering).
- *Total order* conditions, which use a total order $\leq$ on indicators of states. Examples are the number of tokens in a Petri net marking or the number of messages in a specific channel of an LCS.
- *Subset* conditions, which use the subset relation $\subseteq$ on indicators of states. Here, examples are the set of places marked with at least one token for Petri nets or the set of messages in a specific channel of an LCS.

For these classes of conditions, we show how they can be translated into effective and efficient data structures. Therefore, we consider an indicator function f, a necessary condition of the form

$$x \preceq x' \quad \text{implies} \quad f(x) \sim f(x') ,$$

where $\sim$ is one of $\{=, \leq, \subseteq\}$, and discuss how a set X of states can be represented to quickly allow access to equivalence classes of f in X. If we have to test for $x \in \uparrow X$, we only need to test for $x \in \uparrow \{ y \in X \mid y \sim_f x \}$ which considers a potentially smaller set of states than X.
We lift this idea to collect the intersection of X with equivalence classes and we allow relation $\sim$ to be one of $\{=, \leq, \geq, \subseteq, \supseteq\}$. We define the subset $X_{\sim f(x)} \subseteq X$ by

$$X_{\sim f(x)} = \{ y \in X \mid f(y) \sim f(x) \} .$$

This subset $X_{\sim f(x)}$ subsumes the intersection of equivalence classes of indicator function f with X which contain states that can be in WQO relation with x. Only the states in $X_{\sim f(x)}$ have to be explicitly compared to x w.r.t. the WQO and comparison to those states in $X \setminus X_{\sim f(x)}$ is being efficiently omitted.

6.4. Equality Conditions

For an indicator function f and a necessary condition of the form

$$x \preceq x' \quad \text{implies} \quad f(x) = f(x')$$

we need means to retrieve the states in a finite set X that belong to the equivalence class $X_{=f(x)}$. Our approach is to partition X into non-empty subsets of the equivalence classes of f in a way that allows efficient access to partitions with a matching indicator value.

Example 6.2. As an example consider a set of LCS configurations

$$X = \{ (q_1, W_1), (q_1, W_2), (q_2, W_3), (q_3, W_4), (q_3, W_5) \}$$

together with the indicator function that maps a configuration to its control state, $f_{cs}(q, W) := q$. We partition X into three sets according to the value of f_{cs} for each element:

$$X = \underbrace{\{ (q_1, W_1), (q_1, W_2) \}}_{X_{=q_1}} \cup \underbrace{\{ (q_2, W_3) \}}_{X_{=q_2}} \cup \underbrace{\{ (q_3, W_4), (q_3, W_5) \}}_{X_{=q_3}} .$$

If we need to test if $(q_3, W_6) \in \uparrow X$ holds, we find the subset of X with matching value of the indicator function, i.e. the subset $X \cap [(q_3, W_6)]_{f_{cs}} = X_{=f_{cs}(q_3, W_6)}$ of equivalence class $[(q_3, W_6)]_{f_{cs}}$. Thus, we reduced the question if $(q_3, W_6) \in \uparrow X$ holds to the question if (q_3, W_6) is in the upward-closure of $\{ (q_3, W_4), (q_3, W_5) \}$ holds. So fewer states have to be compared. $\Diamond$

Out of the many data structures that allow for efficient access to key-value pairs (indicator value and block) we chose hash tables (cf. for example [Knu98, Sed02]). With hash tables we can efficiently associate an indicator value with a block and find the block for an indicator value. On average, all operations on hash tables only take constant time if the hash function that takes indicator values as input has certain desirable properties such as a small number of colliding hash values.

Assume that the indicator function f takes values in Y and we have a function $H : Y \longrightarrow 2^X$ representing a hash table associating indicator values with subsets of X. The six basic operations we identified in

Sect. 6.2 are easily translated for a hash table implementation (cf. Def. 3.3 on p. 76 for function assignment).

1. Clear
   $X := \emptyset$ is represented by $H(Y) := \emptyset$.

2. Test if empty
   $X \stackrel{?}{=} \emptyset$ is represented by an emptiness test on the hash table.

3. Add an element
   $X := X \cup \{ x \}$ becomes $H(f(x)) := H(f(x)) \cup \{ x \}$.

4. Remove an element
   $X := X \setminus \{ x \}$ becomes $H(f(x)) := H(f(x)) \setminus \{ x \}$.

5. Given state x, remove all elements covering x
   $X := X \setminus \uparrow x$ becomes $H(f(x)) := H(f(x)) \setminus \uparrow x$.

6. Given state x, test if elements covered by x are contained
   $x \stackrel{?}{\in} \uparrow X$ is represented by the test $x \stackrel{?}{\in} \uparrow H(f(x))$.

As mentioned in the previous section, each block of X can be structured using another necessary condition. In Sect. 6.7 we discuss this subject.

6.5. Total Order Conditions

Consider we have an indicator function f and a necessary condition which employs a total order in the form

$$x \preceq x' \quad \text{implies} \quad f(x) \leq f(x').$$

In contrast to equality conditions it does not suffice to find a single block $X_{=f(x)}$ of finite set X which we have to compare against when performing operations like $x \in \uparrow X$. Here, we have to find the set $X_{\leq f(x)}$, which is a collection of blocks with an indicator value that is at most $f(x)$.

Example 6.3. Let X be a set of markings of Petri net N_{ex} (cf. Fig. 2.2 on p. 14)

$$X = \{ p_6, p_7, p_1 + p_2, 2p_3, p_1 + 2p_4, 2p_2 + 2p_4 + p_5 \}$$

and $f_\Sigma : \mathbb{N}^P \longrightarrow \mathbb{N}$ be an indicator function that maps a marking to the sum of its tokens, i.e. $f_\Sigma(m) := \sum_{p \in P} m(p)$. We partition X into blocks according to the indicator function:

$$X = \underbrace{\underbrace{\{ p_6, p_7 \}}_{X_{=1}} \cup \underbrace{\{ p_1 + p_2, 2p_3 \}}_{X_{=2}}}_{X_{\leq 2}} \cup \underbrace{\{ p_1 + 2p_4 \}}_{X_{=3}} \cup \underbrace{\{ 2p_2 + 2p_4 + p_5 \}}_{X_{=5}}$$

Testing if marking $2p_4$ is in the upward-closure of X requires only to test if $2p_4$ is in subset $X_{\leq f_\Sigma(2p_4)} = X_{\leq 2}$, i.e. $X_{=0} \cup X_{=1} \cup X_{=2}$, since every state in a block with a greater indicator value than 2 contains more tokens and violates the necessary total order condition. Thus, a state in $X_{\geq 3}$ cannot be less than or equal to $2p_4$.
There exist similar indicator functions for lossy channel systems. For an LCS with channels C, we can take an indicator function $f_{\#,c}(q, W) = |W(c)|$ that maps a configuration to the number of messages on a specific channel $c \in C$. Of course, another function may return the total number of messages on all channels. $\Diamond$

For this type of condition, we want to store the blocks of a finite set X in an ascending order w.r.t. the total order $\leq$ on indicator values, so that (balanced) search trees, simple arrays, or lists (cf. for example [Knu98, Sed02]) are viable data structures. As we need to go through the list in ascending and in descending order of indicator values, we choose a simple doubly linked list of key-value pairs (indicator values and blocks) for our structure.[2]
Since the access to and modification of doubly linked list is well-understood, we abstract from actual code and describe the necessary actions verbally.

1. Clear
   $X := \emptyset$ is represented by clearing the doubly linked list.

2. Test if empty
   $X \stackrel{?}{=} \emptyset$ is represented by an emptiness test on the list.

[2]Of course, balanced search trees have lower average complexity operations. For the benchmarks we discuss in Sect. 7.2 on p. 207, the number of blocks stayed below a thousand—which is considered “small”—, so that the overhead of maintaining a balanced search tree ruled in favour of a simple list structure.

3. Add an element
   $X := X \cup \{ x \}$ becomes traversing the list in ascending order until either $X_{=f(x)}$ is reached and x is added to that block, or an $X_{=f(x')}$ with $f(x') > f(x)$ or the end of the list is reached. In the latter case, block $X_{=f(x)} = \{ x \}$ is inserted before $X_{=f(x')}$ or appended at the end of the list.

4. Remove an element
   $X := X \setminus \{ x \}$ is achieved by traversing the list in ascending order until either $X_{=f(x)}$ is reached and the element x is removed from that block (if the block is empty, it is removed from the list), or an $X_{=f(x')}$ with $f(x') > f(x)$ or the end of the list is reached. In that case, we are sure that x was not contained in X.

5. Given state x, remove all elements covering x
   $X := X \setminus \uparrow x$ is represented by traversing the list in descending order and removing $\uparrow x$ from each of the partitions in $X_{\geq f(x)}$. The traversal stops when either an $X_{=f(x')}$ with $f(x') < f(x)$ or the beginning of the list is reached.

6. Given state x, test if elements covered by x are contained
   $x \overset{?}{\in} \uparrow X$ is achieved by traversing the list in ascending order and testing for $x \in \uparrow X_{=f(x')}$ for every block with $f(x') \leq f(x)$. If one of the tests returns **true**, the result of the request $x \overset{?}{\in} \uparrow X$ is also **true**. As soon as an $X_{=f(x')}$ with $f(x') > f(x)$ or the end of the list is reached the search stops and the result is **false**.

6.6. Subset Conditions

In the case of subset conditions we have an indicator function f and a necessary condition of the form

$$x \preceq x' \quad \text{implies} \quad f(x) \subseteq f(x')\,,$$

where $f : X \longrightarrow 2^Y$ takes elements of a set X as input and maps it to *finite* sets over some arbitrary codomain Y. As with total order conditions, we delegate the check $x \in \uparrow X$ to a set of blocks $X_{\subseteq f(x)}$ of X, immediately excluding comparisons with states not in $X_{\subseteq f(x)}$.

Example 6.4. Let X be a set of markings of Petri net N_{ex} (cf. Fig. 2.2 on p. 14)

$$X = \{ 3p_2,\, p_1 + 2p_2,\, 3p_1 + p_2,\, 2p_3 + p_5,\, p_3 + 9p_5,\, p_6 \}$$

and $f_{supp} : \mathbb{N}^P \longrightarrow 2^P$ be an indicator function that maps a marking to the set of marked places (*support*), i.e. $f_{supp}(m) := \{ p \in P \mid m(p) \neq 0 \}$. We partition X into the subsets according to the indicator function:

$$X = \underbrace{\underbrace{\{ 3p_2 \}}_{X_{=\{ p_2 \}}} \cup \underbrace{\{ p_1 + 2p_2, 3p_1 + p_2 \}}_{X_{=\{ p_1,p_2 \}}}}_{X_{\subseteq\{ p_1,p_2 \}}} \cup \underbrace{\{ 2p_3 + p_5, p_3 + 9p_5 \}}_{X_{=\{ p_3,p_5 \}}} \cup \underbrace{\{ p_1 + p_6 \}}_{X_{=\{ p_1,p_6 \}}}$$

Testing if marking $2p_1 + p_2$ is in the upward-closure of X requires only to check if $2p_1 + p_2$ is in the subset

$$X_{\subseteq f_{supp}(2p_1+p_2)} = X_{\subseteq\{ p_1,p_2 \}} = X_{=\emptyset} \cup X_{=\{ p_1 \}} \cup X_{=\{ p_2 \}} \cup X_{=\{ p_1,p_2 \}}\,,$$

since every marking in a block with a different indicator value puts tokens on places that are not marked in x and violates the necessary subset condition. Thus, a state in $X \setminus X_{\subseteq\{ p_1,p_2 \}}$ cannot be greater than or equal to $p_1 + p_2$.
There exist similar indicator functions for lossy channel systems. For an LCS with channels C, we can take an indicator function

$$f_{msg,c}(q, W) = \{ m \in M \mid \exists v, w \in M^* : W(c) = v \cdot m \cdot w \}$$

that maps a configuration to the set of messages on a specific channel $c \in C$. Of course, another function may return the set of messages on all channels. ♢

These conditions ask for a data structure that is more involved than simple lists as the blocks that belong to all subsets of an indicator value have to be retrieved efficiently. The data structure we propose to use for this type of necessary condition is a special instantiation of a binary tree.

Definition 6.3 (Binary Tree). A *binary tree* is a connected graph (V, E, v_0) without cycles, consisting of a set of *nodes* V, a set of *edges* $E \subseteq V \times V$ and a root node $v_0 \in V$. A directed edge $(v, v') \in E$ goes from the *parent* node v to the *child* node v'. The root node v_0 is the only

node without a parent and every node has exactly 0, 1, or 2 child nodes. If a node has no children it is called a *leaf* node, else it is an *inner node*. The number of edges in the path from the root node to a node is the *height* of that node. The tree *height* is the maximal node height in the tree. $\diamondsuit$

If the nodes of a binary tree impose some form of ordering on values that are attached to nodes—for example, all “left” descendants are smaller than the current node—, we speak of *binary search trees* (BSTs).
We will introduce our specialized data structure that allows to search a powerset via a BST in two steps: First, we discuss the use of *full* binary trees that have $2^{|Y|} - 1$ internal nodes and $2^{|Y|}$ leaves and then turn to trees that are not necessarily full and can have a considerably less number of internal nodes and leaves and do not require the codomain Y of the indicator function f to be finite. As we use the trees to search through the powerset of Y, we call them *powerset search trees* (PSTs).

6.6.1. Full Powerset Search Trees

In the context of full powerset trees, we constrain the indicator function to have a finite codomain, s.t. $f : Z \longrightarrow 2^Y$ and $X \subseteq Z$ which gives rise to a finite search tree over the powerset of Y.

Definition 6.4 (Full Powerset Search Tree). Let $f : Z \longrightarrow 2^Y$ be a function with Y finite. A *full powerset search tree* for $X \subseteq Z$ over finite set Y is a rooted binary tree $(V, E, v_0, \lambda, \sigma)$ with labelled edges and nodes that are labelled via the two functions λ and σ. Full powerset search trees satisfy the following constraints:

- Every leaf node has height $|Y|$.
- $E \subseteq V \times \{ \bot, \top \} \times V$ is the labelled edge relation. We write $E(v)$ to denote the set $\{ v' \in V \mid (v, q, v') \in E \}$ of child nodes of v. Every inner node has one child with an $\bot$-labelled edge and one child with an $\top$-labelled edge.
- $\lambda : V \longrightarrow (Y \cup \{ \bot \})$ is the labelling function that assigns to each inner node an element from Y and $\bot$ to every leaf node. For every sequence of edges

$$(v_{a_0}, q_0, v_{a_1}), (v_{a_1}, q_1, v_{a_2}), \ldots, (v_{a_{|Y|-1}}, q_{|Y|-1}, v_{a_{|Y|}})$$

from the root node $v_{a_0} = v_0$ to a leaf node $v_{a_{|Y|}}$, the set of visited labels of inner nodes $\bigcup_{0 \leq i < |Y|} \lambda(v_{a_i})$ is exactly the set Y.

- $\sigma : V \longrightarrow \mathcal{P}(X)$ attaches to each node a set of elements from X and the union of all attached sets is X, formally $X = \bigcup_{v \in V} \sigma(v)$. Only leaf nodes have non-empty sets attached: $\sigma(v) \neq \emptyset \Rightarrow E(v) = \emptyset$.

- If a node v that is connected to the root node via edge sequence

$$(v_{a_0}, q_0, v_{a_1}), (v_{a_1}, q_1, v_{a_2}), \ldots, (v_{a_{k-1}}, q_{k-1}, v_{a_k})$$

(with $v_0 = v_{a_0}$ and $v = v_{a_k}$), we define the *characteristic set* of v to be

$$\chi(v) := \{ y \in Y \mid \exists 1 \leq i < k : (\lambda(v_{a_i}) = y \wedge q_i = \top) \} .$$

The set attached to each node by σ is a subset of the block $X_{=\chi(v)}$, formally

$$\forall v \in V : \sigma(v) \subseteq X_{=\chi(v)} ,$$

meaning that for any $x \in \sigma(v)$ the indicator value $f(x)$ of x is exactly the characteristic set $\chi(v)$ of node v. $\Diamond$

The last two items of the definition ensure that all the blocks of X are attached to the leaf nodes that are reached from the root node via a path which describes the value of the indicator function for the elements in that block. As a shorthand, we call a node v' that is connected to its parent via a $\bot$-labelled edge $v, \bot, v'$ a $\bot$-*child*—analogously, $\top$-children are defined. The following example gives a nice visualization of a full powerset search tree.

Example 6.5. Let X be a set of markings of some Petri net with only four places,

$$X = \{ 7p_4,\ p_2 + p_3 + 2p_4,\ p_2 + 2p_3 + p_4,\ p_1 + p_3 + 2p_4,\ p_1 + p_2 + 2p_3 \} ,$$

and $f_{supp} : \mathbb{N}^P \longrightarrow 2^P$ be the indicator function from Example 6.4. We partition X into blocks according to the indicator function and attach

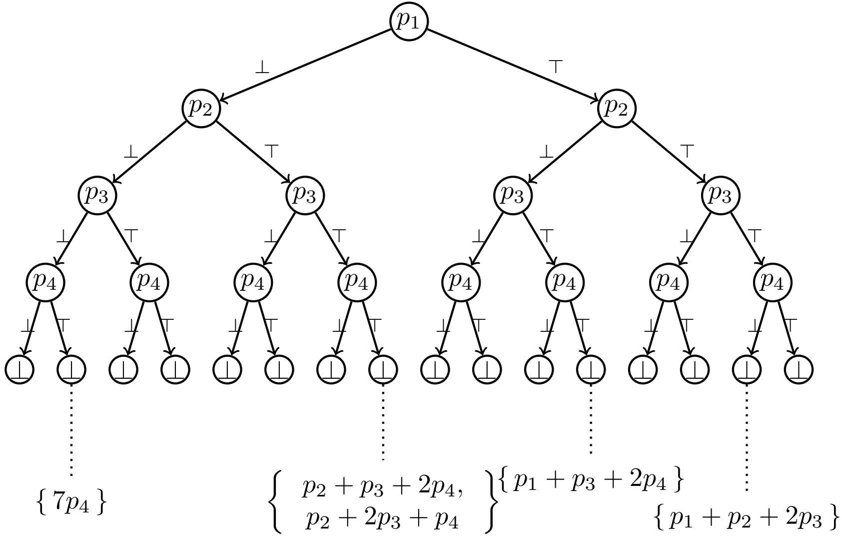


Figure 6.2.: Example of a full powerset search tree.

these blocks to the full powerset search tree presented in Fig. 6.2.[3] We omitted drawing the empty sets attached to every other node by σ.
As we can see, marking $7p_4$ in contained the set attached to a node that is reached (from the root node) via a path where the only $\top$-labelled edge originates from a node labelled with p_4. Therefore the characteristic set of the node is $\{ p_4 \}$, just like the indicator value of $7p_4$: $f_{supp}(7p_4) = \{ p_4 \}$. The two markings $p_2+p_3+2p_4$ and $p_2+2p_3+p_4$ have the same indicator value and therefore are contained in the same set.
In this example we observe that the tree is only sparsely populated in terms of non-empty sets attached to (leaf) nodes. $\diamondsuit$

With this data structure, we are fit to adapt the set of operations for efficient use of subset conditions. Consider a full powerset search tree $(V, E, v_0, \lambda, \sigma)$ for some set X with indicator function $f : Z \longrightarrow 2^Y$, $X \subseteq Z$, and Y finite.

[3]For this tree we chose to have a fixed order in the labelling of the nodes: Each node's label corresponds to its height plus one. However, the definition does not require such a fixed ordering.

1. Clear
   $X := \emptyset$ is represented by clearing out every set of states attached via σ.

2. Test if empty
   $X \stackrel{?}{=} \emptyset$ is represented by an emptiness test on the attached sets. In an actual implementation, an integer variable would be used to keep track of the number of stored elements. The emptiness test would check if that variable is 0.

3. Add an element
   $X := X \cup \{ x \}$ becomes finding the leaf node v with the matching characteristic set $\chi(v) = f(x)$ and updating the function that attaches sets to nodes by $\sigma(v) := \sigma(v) \cup \{ x \}$.

   To find the leaf note, we begin at root node v_0 and test if $\lambda(v_0)$ is an element of $f(x)$ to decide which of the two outgoing edges to take. If $\lambda(v_0) \in f(x)$, then we take edge $(v_0, \top, v')$ to the v'-rooted subtree where each leaf node has $\lambda(v_0)$ in its characteristic set. Else, if $\lambda(v_0) \notin f(x)$, edge $(v_0, \perp v'')$ leads to the v''-rooted subtree where no leaf node has $\lambda(v_0)$ in its characteristic set. We iterate this procedure until we end up at the desired leaf node v.

4. Remove an element
   $X := X \setminus \{ x \}$ becomes finding the leaf node v with the matching characteristic set $\chi(v) = f(x)$ and updating the function that attaches sets to nodes by $\sigma(v) := \sigma(v) \setminus \{ x \}$. The process to find v is the same as to add an element.

5. Given state x, remove all elements covering x
   $X := X \setminus {\uparrow} x$ becomes finding all leaf nodes v_i where the characteristic set $\chi(v_i)$ is a superset of the indicator value of x, i.e. $\chi(v_i) \supseteq f(x)$, and updating the function that attaches sets to nodes by $\sigma(v_i) := \sigma(v_i) \setminus {\uparrow} x$ for any such leaf node.

   The search for all the leaf nodes with suitable characteristic sets differs from the procedure described in the context of adding an element by including *both* subtrees for further search in case the node's label $\lambda(v_0)$ is not contained in $f(x)$. If $\lambda(v_0) \in f(x)$, the search proceeds as usual and only the subtree connected by the

$\top$-labelled edge is searched. This way, the search maintains a set of nodes as the current state and collects all leaf nodes with $\chi(v_i) \supseteq f(x)$.

6. Given state x, test if elements covered by x are contained $x \overset{?}{\in} \uparrow X$ is represented by finding all leaf nodes v_i where the characteristic set $\chi(v_i)$ is a subset of the indicator value of x, i.e. $\chi(v_i) \subseteq f(x)$, testing if state x is in the upward-closure of the set attached to any of these nodes: $x \overset{?}{\in} \uparrow \bigcup_i \sigma(v_i)$. The process to find all the wanted leaf nodes is analogous to the one for removing $\uparrow x$ from X but with the search looking at the subtree connected via the $\bot$-labelled edge if $\lambda(v_0) \notin f(x)$ and the search being expanded to both children in case that $\lambda(v_0) \in f(x)$.

We will see an example of the a involved operation in the upcoming section where we also discuss correctness of our method.
In Example 6.5 we observed that full powerset search trees experience sparsely populated leaves which creates an overhead in memory consumption and running time. We intend to relax the constraints of full powerset search trees in order to allow for superfluous nodes to be left out. The construction of a *relaxed* powerset search tree aligns to the elements that are added to it, meaning that the tree grows subtrees and leaves with certain characteristic sets on demand. By beginning with a smaller tree and extending it upon insertion of new elements, trees with fewer leaves—and more significantly with reduced tree height—are created, which is of great benefit.

6.6.2. Relaxed Powerset Search Trees

For relaxed powerset trees, the indicator function $f : Z \longrightarrow 2^Y$ can have an arbitrary (possibly infinite) codomain Y, as long as f is *image-finite*, i.e. $|f(x)| \in \mathbb{N}$ for every $x \in Z$. The following formal definition of relaxed powerset trees does neither require the tree to have a fixed number of nodes, nor does σ attach non-empty (subsets of) blocks to leaf nodes only.

Definition 6.5 ((Relaxed) Powerset Search Tree). Let $f : Z \longrightarrow 2^Y$ be an image-finite function. A *(relaxed) powerset search tree* for $X \subseteq$

Z over set Y is a binary tree $(V, E, v_0, \lambda, \sigma)$ with labelled nodes and labelled edges satisfying the following constraints:

- $E \subseteq V \times \{\bot, \top\} \times V$ is the labelled edge relation and we write $E(v)$ to denote the set $\{ v' \in V \mid (v, q, v') \in E \}$ of child nodes of v. Every inner node has one child with an $\bot$-labelled edge and one child with an $\top$-labelled edge.

- $\lambda : V \longrightarrow (Y \cup \{ \bot \})$ is the labelling function that assigns to each inner node an element from Y and $\bot$ to every leaf node. For every sequence of edges

  $$(v_{a_0}, q_0, v_{a_1}), (v_{a_1}, q_1, v_{a_2}), \ldots, (v_{a_{k-1}}, q_{k-1}, v_{a_k})$$

  from the root node $v_{a_0} = v_0$ to a leaf node v_{a_k}, the set of visited labels of inner nodes $\bigcup_{0 \leq i < k} \lambda(v_{a_i})$ is a subset of Y.

- $\sigma : V \longrightarrow \mathcal{P}(X)$ attaches to each node a set of elements from X and the union of all attached sets is X, formally $X = \bigcup_{v \in V} \sigma(v)$.

- If a node v that is connected to the root node via edge sequence

  $$(v_{a_0}, q_0, v_{a_1}), (v_{a_1}, q_1, v_{a_2}), \ldots, (v_{a_{k-1}}, q_{k-1}, v_{a_k})$$

  (with $v_0 = v_{a_0}$ and $v = v_{a_k}$), we define the *characteristic set* of v to be

  $$\chi(v) := \{ y \in Y \mid \exists 1 \leq i < k : (\lambda(v_{a_i}) = y \wedge q_i = \top) \} .$$

  The set attached to each node by σ is a subset of the block $X_{=\chi(v)}$, formally

  $$\forall v \in V : \sigma(v) \subseteq X_{=\chi(v)} ,$$

  meaning that for any $x \in \sigma(v)$ the indicator value $f(x)$ of x coincides with the characteristic set $\chi(v)$ of node v. ◊

For sake of brevity, we mean relaxed powerset search trees when we write *powerset search trees.*

In contrast to full powerset search trees, relaxed powerset search trees may have a height lower than $|Y| + 1$ and may be imbalanced. A full powerset search tree thus is a special case of a relaxed powerset search

tree where the tree has the maximal number of nodes and only leaves have non-empty sets attached.

The last two items of the definition ensure that all the blocks of X are attached to *some* nodes that are reached from the root node via a path which describes the value of the indicator function for the elements in that block. A block may be split into several subsets that are attached to different nodes with the same characteristic set. Take for example the (full) powerset search tree from Example 6.5 where the characteristic set of the node reached by the sequences $\top, \top$ (a node labelled with p_3) and $\top, \top, \bot$ (a node labelled with p_4) and $\top, \top, \bot, \bot$ (a node labelled with $\bot$) of edge labels are all the same: $\{ p_1, p_2 \}$. We fix this observation in the following corollary.

Corollary 6.1 (Characteristic Sets of Connected Nodes). Let $f : Z \longrightarrow 2^Y$ be an image-finite function and let $(V, E, v_0, \lambda, \sigma)$ be a relaxed powerset search tree.

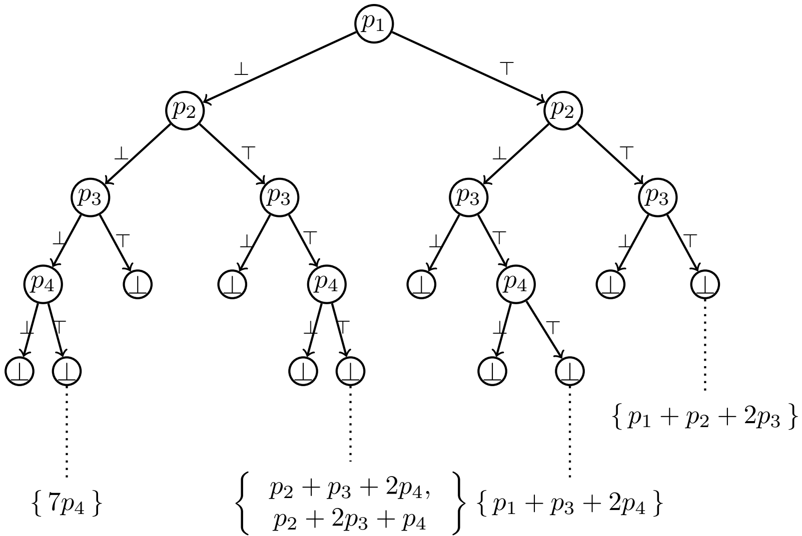
- For any pair of nodes v, v' that is connected via a $\bot$-labelled edge $(v, \bot, v')$, the characteristic sets of the nodes coincide: $\chi(v) = \chi(v')$.
- For any pair of nodes v, v' that is connected via a $\top$-labelled edge $(v, \top, v')$, the characteristic set of the parent node is a subset of the characteristic set of the child node: $\chi(v) \subseteq \chi(v')$.

Proof. Theses properties follow directly from the last item of the definition of characteristic sets for relaxed powerset search trees (Def. 6.5). □

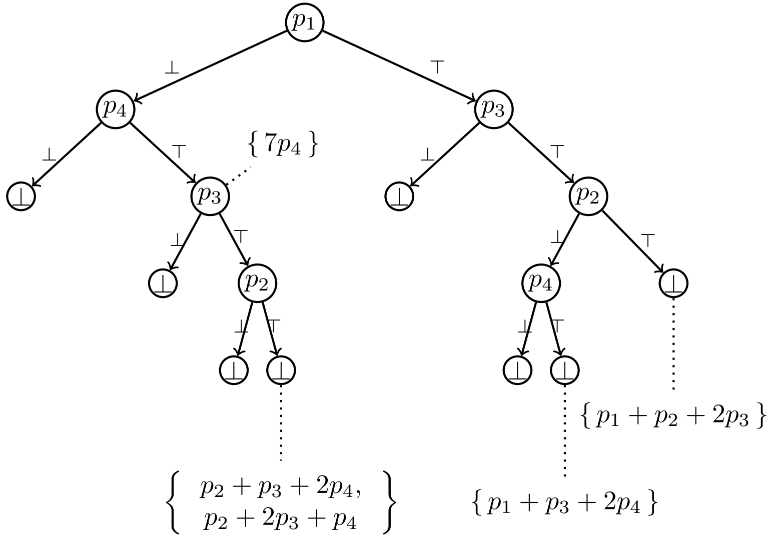
In order to read a block from a relaxed powerset search tree, the union of sets attached to several nodes of the tree may have to be constructed. While the relaxed powerset search tree allows for arbitrary descendant nodes of a node v to have the same label $\lambda(v)$, it is of no benefit to have such a constellation of nodes as it artificially blows up the node height. Furthermore, a relaxed powerset search tree does not need to have a node with a characteristic set for any element of 2^Y.

The following example shows two relaxed powerset search trees representing the same set of states.

Example 6.6. Both trees in Fig. 6.3 show relaxed PST representations of the same set that we introduced in Example 6.5. Again, we omitted drawing the empty sets attached to nodes by σ.



(a) Example of a relaxed powerset search tree I.



(b) Example of a relaxed powerset search tree II.

Figure 6.3.: Examples of relaxed powerset search trees.

As we can see, marking $7p_4$ in contained the set attached to a node that is reached (from the root node) via a path where the only $\top$-labelled edge originates from a node labelled with p_4. The differences between the full PST of Fig. 6.2 and Fig. 6.3a lies in the removal of inner nodes that have only children with empty sets attached and moving the marking $p_1 + p_2 + 2p_3$ that was attached to a $\bot$-child upwards in the tree until a $\top$-child was reached.
The representation of a set of states via PSTs is not unique. Observe that Fig. 6.3a and Fig. 6.3b have a different number of nodes which are differently labelled. Observe that marking $7p_4$ belongs to a set that is attached to an *inner* node with characteristic set $\{ p_4 \}$. ♢

A takeaway of the two trees in Example 6.6 is that there exist orderings of node labels that result in smaller trees than other orderings. As we strive for a rather simple data structure, at this stage we are not interested in restructuring the PST by rotating, merging, or swapping nodes as common in the field of balanced BSTs to reduce the tree height or the number of nodes.
We want to construct a PST for a set step by step and in a lazy fashion. Therefore, we extend a PST by predefined *tree extensions* which can be merged with existing PSTs to form larger PSTs (see Example 6.7 for a visualization).

Definition 6.6 (Tree Extension). Let $f : Z \longrightarrow 2^Y$ be an image-finite indicator function, x an element of Z, and $R = \{ y_1, \ldots, y_k \} \subseteq f(x)$ a subset of the indicator value of x. A *tree extension for x w.r.t. R* is a rooted binary tree $(V, E, v_1^\top, \lambda, \sigma)$ with

- $V := \{ v_1^\top \} \cup \{ v_2^\bot, v_2^\top, v_3^\bot, v_3^\top, \ldots, v_{k+1}^\bot, v_{k+1}^\top \}$,
- $E := \{ (v_1^\top, \bot, v_2^\bot), (v_1^\top, \top, v_2^\top), \ldots, (v_k^\top, \bot, v_{k+1}^\bot), (v_k^\top, \top, v_{k+1}^\top) \}$,
- $\lambda(v_{k+1}^\top) := \lambda(v_i^\bot) := \bot$ for all $1 < i \leq k+1$ and $\lambda(v_i^\top) := y_i$ for all $1 \leq i \leq k$,
- $\sigma(v_{k+1}^\top) := \{ x \}$ and $\sigma(v_i^\top) := \sigma(v_{i+1}^\bot) := \emptyset$ for all $1 \leq i \leq k$. ♢

With the definition of characteristic sets as for PSTs, we see that $\chi(v_k^\top)$ is R. This means that if the root node of a tree extension was connected to a PST node with characteristic set R', the characteristic set of $v_k^\top$ would become $R \cup R'$. To put tree extensions to use, we introduce a specialized

merge operation that connects the root node of a tree extension for x to a node of a PST so that the characteristic set of node $v_k^\top$—where x is stored—is forced to be $f(x)$. This is achieved by requesting the node where the tree extension is merged to have characteristic set $f(x) \setminus R$.

Definition 6.7 (Merge). Let $f : Z \longrightarrow 2^Y$ be an image-finite indicator function, x an element of Z, and $R \subseteq f(x)$ a subset of the indicator value of x. Consider a relaxed PST $P = (V, E, v_0, \lambda, \sigma)$ for X over Y, some leaf node $v \in V$ with $\lambda(v) = \bot$ and characteristic set $\chi(v) = f(x) \setminus R$, and a tree extension $Q = (V^e, E^e, v_1^\top, \lambda^e, \sigma^e)$ for x w.r.t. R.
The result of *merging* P and Q at v is the binary tree $(V', E', v_0', \lambda', \sigma')$ with

- $V' := (V \setminus \{ v \}) \cup V^e$,
- E' is E^e if v is the root node v_0 of P, else, if there exists an edge $(v', q, v) \in E$ in P, it is $(E \setminus \{ (v', q, v) \}) \cup E^e \cup \{ (v', q, v_1^\top) \}$,
- v_0' is $v_1^\top$ if v is the root node of P, else v_0' is v_0,
- $\lambda'(w) := \begin{cases} \lambda(w), & \text{if } w \in V \\ \lambda^e(w), & \text{else} \end{cases}$, and
- $\sigma'(w) := \begin{cases} \sigma(v) \cup \sigma^e(v_1^\top), & \text{if } w = v_1^\top \\ \sigma(w), & \text{if } w \in V \setminus \{ v \} \\ \sigma^e(w), & \text{else} \end{cases}$.

Thus, the tree extension replaces node v in the PST and the set attached to v is added to the root node of the tree extension. $\diamondsuit$

As we can see, the merge operation is a simple operation on graphs that can also be understood as a relabelling of node v and appending a leaf and a subtree to that node. In Example 6.7, we apply our proposed mechanism to extend a powerset search tree and show a graphical representation of a tree extension.

Example 6.7. In this example, we extend the PST from Fig. 6.3b by adding a marking $2p_2 + p_3$ for which no node with matching characteristic set exists. Figure 6.4 shows a tree extension (marked gray) for the marking w.r.t. set $\{ p_2, p_3 \}$ and where it can be added to the PST. By

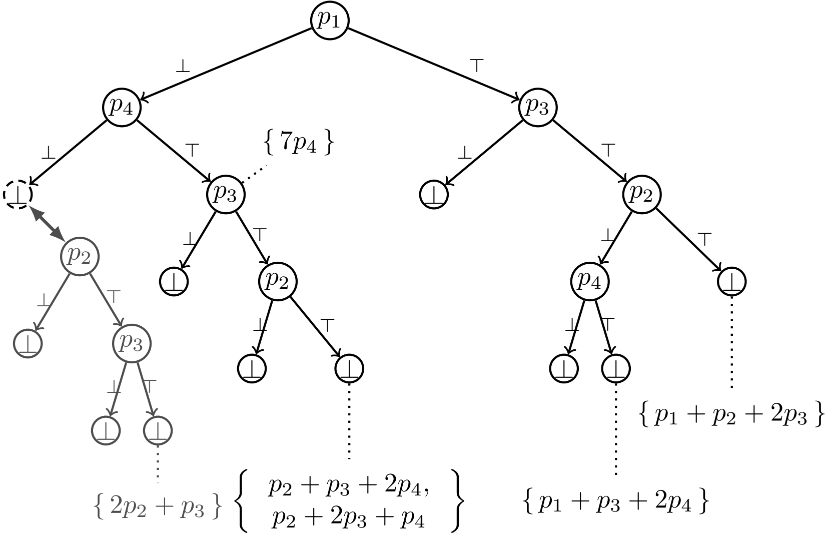


Figure 6.4.: Merging a tree extension with a PST.

replacing the dashed leaf node by the root node of the tree extension—and adding the set attached to the leaf node to the set attached to the extension's root node—a merge operation is carried out. The PST properties are maintained and marking $2p_2 + p_3$ is added to the tree. $\Diamond$

The following lemma shows that the result of the merge operation indeed is an extended powerset search tree.

Lemma 6.2 (Effect of Merge). Merging a PST for X and a tree extension for x results in a PST for $X \cup \{ x \}$.

Proof. Let $f : Z \longrightarrow 2^Y$ be an image-finite indicator function, x an element of Z, and $R \subseteq f(x)$ a subset of the indicator value of x. Consider a relaxed PST $P = (V, E, v_0, \lambda, \sigma)$ for X over Y, some leaf node $v \in V$ with $\lambda(v) = \perp$ and characteristic set $\chi(v) \subseteq R$, and a tree extension $Q = (V^e, E^e, v_1^\top, \lambda^e, \sigma^e)$. Let $P' = (V', E', v_0', \lambda', \sigma')$ be the result of the merge operation of P and Q.
As the nodes v (in P) and $v_1^\top$ (in P') have the same characteristic sets and by Corollary 6.1 (characteristic sets of connected nodes), we have

that by merging P and Q, the characteristic sets of the nodes in tree extension Q are enlarged by $\chi(v)$ (now in P'). Thus, the characteristic set of the leaf node that x is attached to is $f(x)$ and the PST properties are retained. Since the contents of the set attached to v, i.e. $\sigma(v)$, are transferred into $\sigma(v_1^\top)$ and the only element attached to a node of the tree extension is x, the resulting tree is a relaxed powerset search tree for $X \cup \{ x \}$ w.r.t. indicator function f. $\square$

We are now fit to translate the abstract operations of Sect. 6.2 to the context of PSTs. Consider a powerset search tree $(V, E, v_0, \lambda, \sigma)$ for some set X with image-finite indicator function $f : Z \longrightarrow 2^Y$ with $X \subseteq Z$.

1. Clear
   $X := \emptyset$ is represented by clearing out the sets of states attached via σ, i.e. $\sigma(V) := \emptyset$

2. Test if empty
   $X \stackrel{?}{=} \emptyset$ is represented by an emptiness test on the attached sets. As stated for full PSTs: In an actual implementation, a counter variable for the number of contained elements would be used and tested if it is 0.

3. Add an element
   $X := X \cup \{ x \}$ becomes finding some node v with a characteristic set that matches the indicator value, i.e. $\chi(v) = f(x)$, and adding x to the attached set: $\sigma(v) := \sigma(v) \cup \{ x \}$. If no such node exists, the tree is extended at v with a suitable tree extension for x w.r.t. $f(x) \setminus \chi(x)$.

   Formally, a leaf node v with a characteristic set that is a subset of $f(x)$ is found analogous to the search described in the add operation for full PSTs. If $\chi(v) = f(x)$, then the state is added to $\sigma(v)$, else the difference $f(x) \setminus \chi(v)$ is non-empty. In case x was not added to $\sigma(v)$, we merge (Def. 6.7) the tree extension (Def. 6.6) for x w.r.t. $f(x) \setminus \chi(v)$ with the PST at node v. The result is a PST with states $X \cup \{ x \}$ as shown in Lemma 6.2.

4. Remove an element
   $X := X \setminus \{ x \}$ is realized analogously to the operation for full PSTs with the difference being that the attached sets of all nodes

v with $\chi(v) = f(x)$ that lie on the traversed path to a leaf node are updated.

5. Given state x, remove all elements covering x
$X := X \setminus \uparrow x$ too is realized in analogy to the operation for a full PST with the difference being that the attached sets of all nodes v with $\chi(v) \supseteq f(x)$ that are visited are updated.

6. Given state x, test if elements covered by x are contained
$x \overset{?}{\in} \uparrow X$ is realized analogously to the method for full PSTs with the difference being that the attached sets of all nodes v with $\chi(v) \subseteq f(x)$ that are visited are tested.

As mentioned before, we are interested in a simple data structure that performs reasonably efficiently. Despite the fact that certain orderings of node labels are preferable over others, we choose not to (*globally*) rearrange the tree upon insertion or deletion of states, but to only perform *local* changes. In our experiments (cf. Sect. 7.2 on p. 207), we observed that even when a set attached to a leaf node is emptied by a remove operation, it is sensible to leave the tree structure unchanged as it was common that other states with the same indicator value were later added to the tree. If the tree would be kept minimal at every time, removed subtrees are likely to be added back to the tree in a later iteration of the framework algorithm. Based on our experience from the experiments, we decided to avoid further pursuit for minimality of PSTs—which leaves room for future work.

6.6.3. Preliminary Experiments

To practically test the effect on performance by using a tree structure to exploit subset conditions, we ran a set of benchmarks. We wanted to see if partitioning the sets of Petri net (with transfer) markings by the indicator function f_{supp} of Example 6.4 had a positive effect on the running time. Therefore, we deviated a bit from our definition of PSTs and limited the number of blocks, a set of markings could be divided into by setting a bound on the height of the search tree. A tree with height 1 would divide a set into two blocks, a height of 2 would result in up to four blocks, etc. We hoped to observe an exponential drop in running

time with increasing tree height as the number of markings to examine should shrink the more blocks the set was divided into.

We carried out a small[4] benchmark test with different bounds on the height of the relaxed powerset search trees for one of the more involved problems, Petri net delegatebuffer.16.1 with 6503 target markings (cf. Table 7.1 on p. 209 and Fig. 7.7 on p. 222), with an older version of PETRUCHIO/BW. The results, depicted in Fig. 6.5, clearly show the anticipated exponential reduction in average running time and standard deviation. With the benchmark Petri net containing 52 places, the tree height cannot exceed this number. Benchmarks have been executed for trees with k-bounded height, $1 \leq k \leq 52$, as well as the unbounded case, iterated 300 times each to give comparable average running times and standard deviations. The lines "Avg. no. of markings created" and "Avg. no. of markings examined" nearly parallel to the abscissa show that for each bound on tree height the number of objects to be stored in the data structure was about the same.

The standard deviation for "Avg. no. of markings examined" lies around 650 which is a value that is to small for the error bars to be visible in the figure. For the whole benchmark set, the maximal height of trees experienced was 39. The average maximal height was 31.93—not counting those iterations where the maximal height of the tree reached the bound on the tree height. The fact that the average maximal height is much smaller than the theoretical limit of 52 for the tree height reinforces our decision to create powerset search trees on demand and only w.r.t. the elements in the concrete indicator sets of states.

6.7. Hierarchy of Necessary Conditions

For the basic set of necessary conditions we identified—equality conditions, total order conditions, and subset conditions—, we have created a simple construction kit to hierarchically structure large sets of states. To employ these building blocks, a user of our framework inputs a sequence of indicator functions together with the corresponding relation and immediately is presented a data structure. Take Petri nets for example, where a data structure according to the method presented in this

[4]It took 6 days and 18 hours to complete the benchmark.

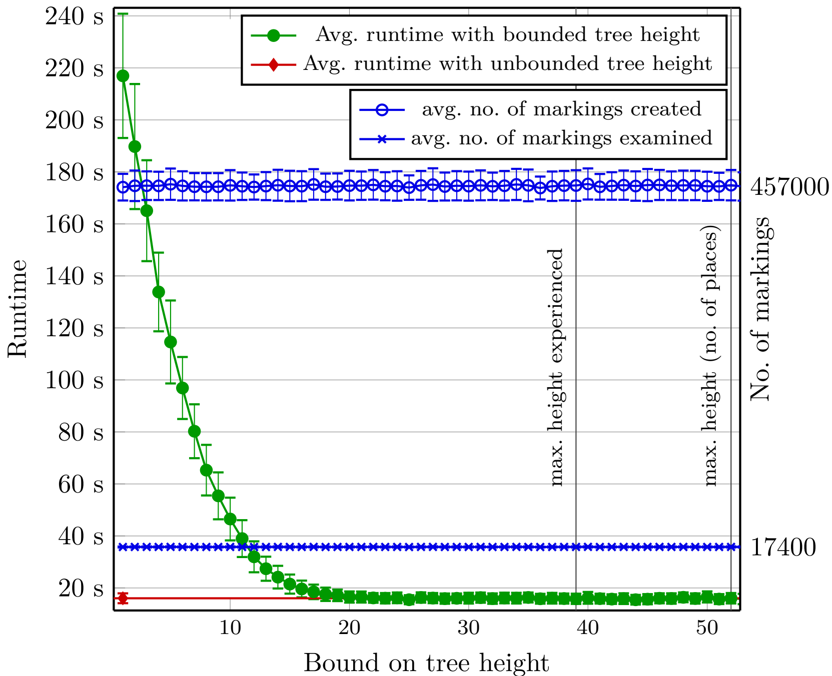


Figure 6.5.: Runtime benefit of partitioning via subset conditions.

chapter can be characterized by the sequence $(f_\Sigma, \leq), (f_{supp}, \subseteq)$, meaning that a set of states will be partitioned via indicator function f_Σ (cf. Example 6.3) which gives rise to a total order condition and the blocks of the set w.r.t. f_Σ are then partitioned via indicator function f_{supp} (cf. Example 6.4) which occurs in a subset condition. However, more complex hierarchies that depend on the concrete model under consideration are possible. As an example, consider an LCS with channel set $C = \{a, b\}$. Here, we could have sequence $(f_{\#,a}, \leq), (f_{\#,b}, \leq), (f_{msg,a}, \subseteq), (f_{msg,b}, \subseteq)$ as the characterization of a data structure that takes into account the specific set of channels in the system.

Since the data structures can be defined rather easily by indicator func-

tions with corresponding relations, the hierarchical order of conditions is something a user may play with. In the following Sect. 6.7.1, we have experimented with different data structures for Petri nets (with transfer) in order to find out whether ordering $(f_\Sigma, \leq), (f_{supp}, \subseteq)$ is preferable over $(f_{supp}, \subseteq), (f_\Sigma, \leq)$.

We want to stress that the idea of structuring a set of states by necessary conditions is a highly extensible framework in itself. Furthermore, the data structures we employ for total order conditions and subset conditions open the doors for parallel processing: In the case of a test if $x \in \uparrow X$ holds for example, states of several different blocks may have to be compared to x. The data structures guarantee these blocks to be disjoint and therefore—in theory—processing them in parallel can be achieved without further precautions. Since there may be very large number of blocks, the overhead of spawning new threads/processes has to be considered when looking for a sweet-spot for the number of parallel processes. Investigation on how the operations on the data structures can be efficiently parallelized to increase the performance is future work.

6.7.1. Preliminary Experiments

In Fig. 6.6 the results of a comparison of both orderings of $(f_\Sigma, \leq)$ and $(f_{supp}, \subseteq)$ are shown.[5] For this benchmark only a subset of models from the following Sect. 7.2 on p. 207 has been chosen as a control sample. Two models—where an early version of Petruchio/BW took considerable time to complete the model checking task—stand out: delegate-buffer.16.1, where a trace is found rather quickly and few markings are created and examined, and HTS , for which fewer markings are created but more markings are examined. The rest of the benchmark set was chosen so that from each level of hardness (solvable in under 0.1 ms, in roughly 0.1 ms, 1 ms, 10 ms, 100 ms, and roughly in 1 s) at least two representative models were present, where at least one marking was to store in the data structure.

For each pair of benchmarks on a model, the number of iterations was the same. The solid bars show average running times where the minimal basis was first divided into blocks w.r.t. f_Σ and these classes were represented by PSTs. Crosshatched bars indicate average running times where the

[5]The benchmark set took 2 days and 6 hours to complete.

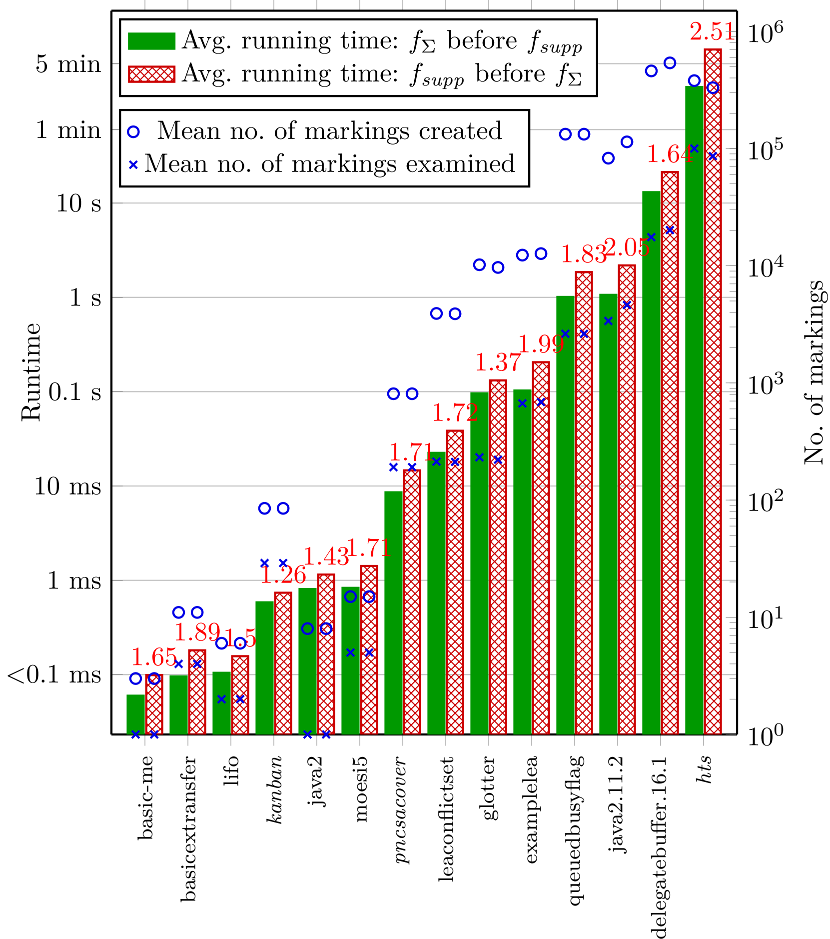


Figure 6.6.: Comparison of hierarchies of total order and subset conditions.

minimal basis was represented by a PST and the blocks w.r.t. f_{supp} at its leaves were further divided into blocks w.r.t. f_{Σ}. Numbers above the bars give the factor of slow down in running time of "crosshatched" approach vs. "solid" approach. Note the logarithmic scales of both running time and number of markings. Clearly, the "solid" approach outperforms the "crosshatched" counterpart and scales well.

6.8. Select Operation

One of the operations we identified in Sect. 6.2 is the selection and removal

$$x := select(W);\ W := W \setminus \{\, x \,\}$$

of a state minimal w.r.t. a total ordering as described in Sect. 5.5 on p. 155. Observe that *select* is the fundamental operation for guiding the search. In particular, the total order on states for search guidance does *not* depend on the well-quasi ordering on states. Therefore, we have to consider two non-related orderings when defining a data structure for sets of states.

In our opinion, there are two apparent options to store a set of states in a way that allows for efficient retrieval of the smallest state w.r.t. one ordering as well as to quickly find states that are comparable to a given state w.r.t. a different ordering:

1. Find a clever method to store information on the ordering of the contained elements for each of the data structures described for equality conditions, total order conditions, and subset conditions. In addition, force every user who extends the construction kit with a new necessary condition and data structure to also find such a method for her data structure.

2. Store a copy of the set of states as an ordered set representation. Use efficient well-understood (standard) data structures.

We decided to go with the second option where we can choose between different data structures for priority queues, for example red-black trees, AVL trees, binary heaps, etc. (see [Knu98, Sed02] for more information). The method is simple and sufficient.

While storing the set W in two different data structures comes at the cost of memory, our benchmarks in the next chapter showed that our reference implementation did not exceed the memory limit of our benchmark system—even for case studies for which another tool struggled.

CHAPTER SEVEN

Reference Implementation and Experiments

Architecture starts when you carefully put two bricks together. — Ludwig Mies van der Rohe, Architect

Contents

7.1 Reference Implementation BW 192
  7.1.1 Architecture 193
  7.1.2 Extensibility 196
  7.1.3 Unit Tests 204
  7.1.4 Usage: An Example 206
7.2 Experiments 207
  7.2.1 Lossy Channel Systems 208
  7.2.2 PN and PNT Benchmark Models 208
  7.2.3 Experiments on SSCs and Search Guidance . 211
  7.2.4 Tool Comparison 214

The algorithmic framework we present in this work is implemented as an extensible, object-oriented Java package together with the general backward acceleration search space construction (SSC), search guides and data structures. In Sect. 7.1, we show how easily it can be extended using the example of Petri nets. Furthermore, we discuss design decisions and show the output of an example run of the program.

In Sect. 7.2, the second part of this chapter, we turn to an extensive experimental evaluation of the performance of our reference implementation on several case studies and compare it with the performance of other tools.

7.1. Reference Implementation BW

With the reference implementation—called BW—we describe in this section, we aim to provide a Java framework that is simple to use and easy to extend.[1] It is geared towards prototyping coverability checkers for new system classes and experimenting with novel optimizations for the backward reachability analysis (BR) via the use of SSCs, without having to "re-implement the wheel".

We give an overview of our implementation, discuss design decisions, and show our process of testing the framework.[2] In this section we demonstrate what is necessary to instantiate it to solve coverability problems for a class of systems using the example of Petri nets (PNs).

Remark 7.1 (Naming of Classes for Data Structures). In the context of our reference implementation, we call the data structures presented in Ch. 6 on p. 160 *(data) abstractions* as the indicator functions abstract from concrete states. For example, the class `SupportAbstraction` represents a set of Petri net markings by the use of a powerset search tree (cf. Sect. 6.6 on p. 170) and a corresponding indicator function.

Design Goals and Performance. Reference implementations are designed with code clarity, encapsulation, and levels of abstraction in mind

[1] It is available at `http://csd.informatik.uni-oldenburg.de/~critter/bw.tar.gz`.

[2] In this work, we will discuss only few of the implementation's 26 000 non-comment lines of code.

to enhance extensibility; performance is not the main concern—aside from efficient algorithms.

Besides the reference implementation, we have implemented the methods presented in this work in the software tool Petruchio/BW with performance and flexibility as the main goals. It was developed as a part of Petruchio [MS10] where it serves as the back-end coverability checker that is needed to compute Meyer's structural semantics of π-calculus processes [Mey08, Mey09] as described in [Str07]. Petruchio/BW is implemented in a more compact fashion, leaving out abstraction levels and disregarding extensibility. The tool represents our previous attempt to implement a coverability checker for the specific system class of Petri nets and grew over the years and in communication with A. Kaiser to support Petri nets with transfers.[3] Experimental results are discussed in Sect. 7.2, together with a comparison to the reference implementation and the tools MIST2[4] [GRV08] and BFC[5] [KKW12].

7.1.1. Architecture

We chose to create the reference implementation using a multi-layered architecture where we have an abstract base layer of interdependent Java interfaces which determine operations (Java methods) of entities (Java objects) visible to the public. Any implementation may choose to have a larger set of operations publicly available but by the use of interfaces, it is treated as a black box w.r.t. the interfaces implemented. By relying on a defined set of interfaces, we allow for all the concrete implementations we provide to be substituted by programs written by a user.

In the example of Listing 7.1, a class that represents a state of a well-structured labelled transition system (WSLTS) has to implement the `State` interface which forces the concrete class to provide the public methods `setTrace(...)` and `getTrace()`. Moreover, as the `State` interface *extends* the `SimpleState` interface, a concrete class also has to implement the operations required by that interface, i.e. a set of comparisons. As the `Transition` interface does not require any visible opera-

[3] A. Kaiser used the MIST2 tool prior to switching to Petruchio/BW for the speed benefit. He later built his own program BFC that is optimized for the models he encountered in his work.

[4] `https://github.com/pierreganty/mist/`

[5] `http://www.cprover.org/bfc/`

tions, it is a so-called flag interface.[6] If it is used, the implementing class is "flagged" to be a transition. Since our framework algorithm does not rely on properties of transitions, a flag interface suffices: we only have to know what objects are transitions.

Listing 7.1: Interfaces for states and transitions

```
1  package bw.interfaces;
2
3  public interface Transition<T extends Transition<T>> { }
4
5  public interface SimpleState<S extends SimpleState<S>> {
6    boolean isLessThan(S s);
7    boolean isGreaterThan(S s);
8    boolean isLessEqualThan(S s);
9    boolean isGreaterEqualThan(S s);
10   boolean isEqualTo(S s);
11 }
12
13 public interface State<S extends State<S, T>,
14     T extends Transition<T>> extends SimpleState<S> {
15   void setTrace(Trace<T> trace);
16   Trace<T> getTrace();
17 }
```

Besides the interfaces that are used to define WSLTSs, implementations of search space constructions, different forms of upward-closed sets (UCSs) and downward-closed sets (DCSs), as well as the analysis that captures our framework algorithm are hidden behind interfaces.

Generics. We decided to heavily rely on the Java generics facility that allows for programs that are type-safe at compile-time.[7]

Package and Folder Structure. The reference implementation BW consists of the Java package `bw` for the source code and some additional folders containing libraries, configuration files, shell scripts, and a few examples. The package is structured as follows.

[6] In our reference implementation, the transitions are labelled intrinsically and we distinguish them in the analysis by their object identity.

[7] Confer Sect. A.1 on p. 235 for more information on generics.

- `bw`: The main package with the executable Java class `Runner`.
  - `interfaces`: Interfaces for WSLTSs, SSCs, parsers, main data structures, and analysis classes.
  - `analysis`: Implementation of our framework algorithm (cf. Sect. 3.3 on p. 79).
  - `base`: Abstract classes for states and models that contain default implementations of most of the interface methods.
    - `cs`: Different implementations of data structures for (downward- and) upward-closed sets based on some abstract structuring of sets of states, inter alia, that allow for the ordering of states as discussed in Sect. 6.8 on p. 189.
      - `abstractions`: Various implementations of data structures induced by generic necessary conditions to structure sets of states as described in Ch. 6 on p. 160.
      - `interfaces`: Interfaces for the data structures in the `cs` package.
    - `ssc`: Implementations of the backward acceleration SSC (as described in Sect. 5.1 on p. 131) and a "chaining" operation on SSCs in relation to Sect. 5.4 on p. 152.
  - `util`: Utility classes for formatting, logging, etc.
    - `concurrent`: Utility classes to simplify the implementation of concurrent operations on data structures using `ExecutorServices` (cf. [Lea99, Blo08]).
  - `benchmark`: Code to repeatedly solve coverability problems and perform some statistical analysis over the resulting data.
  - `impl`: Contains subpackages for different implementations of well-structured labelled transition systems.
    - `cfg`: Instantiation of the framework for context-free grammars (CFGs).
    - `lcs`: Instantiation of the framework for lossy channel systems.
    - `pn`: Instantiation of the framework for Petri nets.
    - `pnt`: Instantiation of the framework for Petri nets with transfers (PNTs).

Furthermore, every subpackage of `bw` contains a package `test` with JUNIT-code to run tests on the classes implemented in that package (if any).

7.1.2. Extensibility

As mentioned in the previous section, the reference implementation BW contains instantiations for PNs, PNTs, LCSs, and CFGs. These were created by implementing the interfaces for 1. states, 2. transitions, 3. models, 4. data abstractions, and 5. parsers. In this section, we focus on a single class of WSLTSs and show that the time and effort needed to extend our reference implementation by the class of Petri nets is rather small.
To relieve the user from implementing non-essential operations—meaning operations that can be derived from a set of core operations—, we have abstract classes that implement most methods of the above interfaces. For example, the interface `bw.interfaces.State` of a WSLTS state (cf. Listing 7.1) asks for the implementation of many comparison operations (for convenience of a user of that interface) and methods to get and set transition sequences (seven in total). When implementing the abstract class `bw.base.State` for the Petri net case, we only have to code the two methods shown in Listing 7.2[8]: a single comparison operation and a method to retrieve a hash value for the state to allow for the use of hash tables as needed for our equality conditions (cf. Sect. 6.4 on p. 167). The necessity to implement the latter method stems from the abstract state, not the interface.

Listing 7.2: Actual methods to implement for a state

```
1  package bw.impl.pn;
2  public class Marking extends bw.base.State<Marking, Transition> {
3      @Override
4      public boolean isLessEqualThan(Marking s) { ... }

6      @Override
7      public int hashCode() { ... }

9      // ... (remainder of class omitted)
10 }
```

[8]The Java annotation `@Override` ensures that the method actually overrides an inherited method.

The interface for models, shown in Listing 7.3, requires the large number of 19 methods to be implemented. However, many of these methods are optional and some of them are even allowed to throw an `UnsupportedOperationException`, indicating that they are not implemented (which methods may do so is documented in the actual source file). Again, an abstract implementation `bw.base.Model` exists that reduces the number of methods that have to be implemented to built a meaningful model to just four. We demonstrate this on our running example of Petri nets in Listing 7.4: The methods to implement fulfill the following purposes of 1. declaring the finite set of initial states of the WSLTS, 2. declaring the finite basis of final states of the WSLTS, 3. choose a set of transitions which are to be used in the computation of covering predecessors of a state, and of 4. computing the set of covering predecessors of a state and w.r.t. a transition.

Listing 7.3: Interface for a model

```
1  package bw.interfaces;
2  import java.util.Collection;
3  import java.util.Comparator;
4  import java.util.List;
5  import bw.analysis.AnalysisState;
6  public interface Model<S extends State<S, T>,
7      T extends Transition<T>> {
8    void getBackwardEnabledTransitions(S state, Collection<T>
         bucket);
9    void getOptimizedPredecessors(S state, T transition,
10       AnalysisState<S, T> anaState, Collection<S> bucket);
11   void getPredecessors(S state, T transition, Collection<S>
         bucket);
12   Collection<S> executeTrace(S state, Trace<T> trace);
13   Comparator<S> getSearchGuide();
14   boolean prune(S state);
15   IterableDcs<S, T> getInitialStates();
16   Iterable<S> getFinalStates();
17   Ucs<S, T> newUcs();
18   Dcs<S, T> newDcs();
19   IterableDcs<S, T> newIterableDcs();
20   SortedUcs<S, T> newSortedUcs();
21   SortedDcs<S, T> newSortedDcs();
22   Trace<T> newTrace(T transition, Trace<T> suffix);
```

```
23   void initializeBeforeModelChecking();
24   void finalizeAfterModelChecking();
25   void handleArgs(List<String> args);
26   Model<S, T> shuffle();
27   String usage();
28 }
```

Listing 7.4: Actual methods to implement for a model

```
1  package bw.impl.pn;
2  import java.util.Collection;
3  import bw.interfaces.IterableDcs;

5  public class PetriNet
6      extends bw.base.Model<Marking, Transition> {
7    @Override
8    public IterableDcs<Marking, Transition> getInitialStates() {
         ... }

10   @Override
11   public Iterable<Marking> getFinalStates() { ... }

13   @Override
14   public void getBackwardEnabledTransitions(Marking state,
15       Collection<Transition> bucket) { ... }

17   @Override
18   public void getPredecessors(Marking state, Transition
         transition,
19       Collection<Marking> bucket) { ... }

21   // ... (remainder of class omitted)
22 }
```

Data Structures for UCSs. In order to employ the data structures from Ch. 6 on p. 160, subclasses of `bw.base.cs.EqualityAbstraction`, `bw.base.cs.TotalOrderAbstraction`, or `bw.base.cs.SubsetAbstraction` are created.
For the case of Petri nets—our running example—, we show how two necessary conditions can be implemented to exploit our predefined data

structures. The schematic UML class diagram[9] of Fig. 7.1 depicts the dependencies between the classes we will discuss. The diagram shows that to employ our data structures in the analysis, the user derives her model class `PetriNet` (in the package `bw.impl.pn`) from the abstract `bw.base.Model` and provides the implementations `TokenSumAbstraction` and `SupportAbstraction` of the corresponding abstract classes by defining the indicator functions (and a total order). One of the abstract classes, namely the `SupportAbstraction`, contains an implementation of our powerset search trees from Sect. 6.6 on p. 170. Later in this section, we comment on the user supplied classes that we omitted from the diagram.

We begin by showing most of the class `TokenSumAbstraction` in Listing 7.5. It uses the indicator function f_Σ from Example 6.3 on p. 168, meaning that it maps a PN marking to an integer value and uses the basic integer comparison which is implemented for `Integer` objects via the `compareTo(...)` method. In our implementation of PN markings, the total sum of tokens in that marking can be accessed via the method `getTokenSum()` of a `Marking` object.

Listing 7.5: PN TokenSumAbstraction

```
1  package bw.impl.pn;
2  import bw.base.cs.abstractions.TotalOrderAbstraction;

4  public class TokenSumAbstraction
5       extends TotalOrderAbstraction<Marking, Transition, Integer> {
6    // ... (constructor omitted)

8    @Override
9    protected Integer getIndicator(final Marking state) {
10     return state.getTokenSum();
11   }

13   @Override
14   protected int compare(final Integer a, final Integer b) {
15     return a.compareTo(b);
16   }
17 }
```

[9] For an introduction to the unified modeling language (UML) see [PP05] for example.

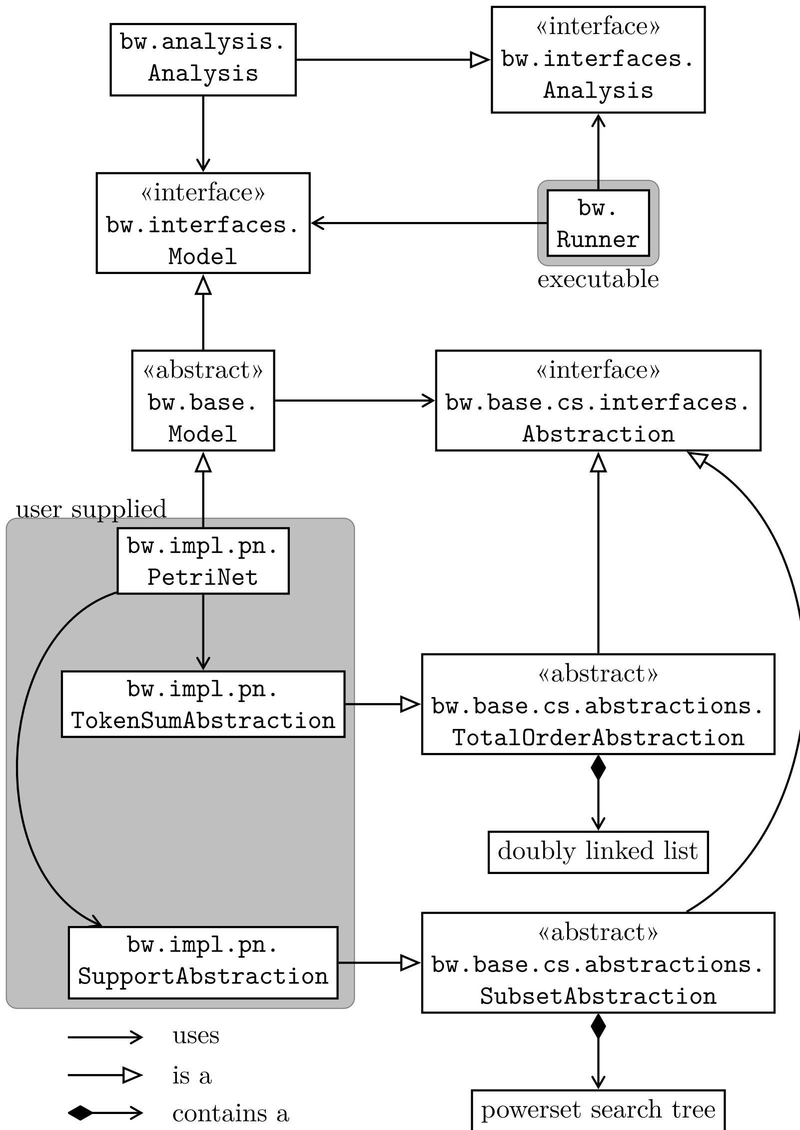


Figure 7.1.: Dependencies between classes for data structures.

The subset condition for Petri nets is even easier to utilize since the comparison is fixed to be the subset relation $\subseteq$. In Listing 7.6 the implementation of such a condition based on indicator function f_{supp} from Example 6.4 on p. 171 is shown. As the indicator function takes values in sets of PN places, the class `SupportAbstraction` expects the `getSet(...)` method to return a set of places for a given marking.

Listing 7.6: PN SupportAbstraction

```
1  package bw.impl.pn;
2  import java.util.Set;
3  import bw.base.cs.abstractions.SubsetAbstraction;

5  public class SupportAbstraction
6        extends SubsetAbstraction<Marking, Transition, Place> {
7    // ... (constructor omitted)

9    @Override
10   protected Set<Place> getSet(final Marking state) {
11     return state.getSupport();
12   }
13 }
```

The brevity of the two classes `TokenSumAbstraction` and `SupportAbstraction` highlights the convenience of using our reference implementation. In our experience from the supported system classes, finding suitable necessary conditions for the well-quasi ordering of states is straightforward. Putting them to use with our pre-assembled data structures comes virtually for free.
With these two data abstractions instantiated, they remain to be put into a hierarchy (cf. Sect. 6.7 on p. 185). In Listing 7.7, the code that expresses the hierarchy for our implementation of Petri nets is shown.

Listing 7.7: PN abstraction hierarchy

```
1  switch(level) {
2    case 1 : return new TokenSumAbstraction(...);
3    case 2 : return new SupportAbstraction(...);
4    case 3 : return new NoAbstraction<Marking, Transition>(...);
5    default : throw new IllegalArgumentException(
6              "No abstraction for level " + level + ".");
7  }
```

It states that for the first level, the `TokenSumAbstraction` is used, the second level is formed by the `SupportAbstraction`, and the last level is a basic collection of states (a doubly linked list to be exact) which we have implemented in the class `NoAbstraction`.

In addition to the three levels for which abstractions (resp, `NoAbstraction`) are defined, in the `default` case an exception is thrown.

Search Guide. To employ a search guide as discussed in Sect. 5.5 on p. 155, the user provides an implementation of the standard `Comparator<X>` interface that represents a total order via the interface method `compare(X x1, X x2)`. An instance of the comparator has to be returned by the method `getSearchGuide()` of the model.

Model Parser. We provide a unified starter `bw.Runner` to execute the analysis for models of different system classes. Therefore, we require users to implement the parser interface shown in Listing 7.8. It constructs a model from a textual representation (read from the `InputStream`) and returns it.

Listing 7.8: Parser interface

```
1 package bw.interfaces;
2 import java.io.InputStream;

4 public interface Parser<S extends State<S, T>,
5     T extends Transition<T>> {
6   Model<S, T> parse(InputStream in);
7 }
```

Concluding Remarks on the Implementation for Petri Nets. To summarize the prototypical instantiation of our framework for Petri nets in our reference implementation, the `bw.impl.pn` package contains the following classes that are visible to the reference implementation.

`Transition` which extends `bw.base.Transition<Transition>`. It contains data and methods that are used by the `PetriNet` class to compute covering predecessors.

Marking which extends bw.base.State<Marking, Transition>. It allows for fast comparison between markings w.r.t. the well-quasi ordering. Furthermore, it contains methods to compute indicator values w.r.t. f_{Σ} and f_{supp}.

Place is the type of the elements in the indicator values for the SupportAbstraction. Like the Transition class, it contains methods that simplify the covering predecessor computation.

TokenSumAbstraction is depicted in Listing 7.5 and it extends bw.base.cs.TotalOrderAbstraction<Marking, Transition, Integer>. The listing omits only the constructor which passes its parameters to the super class.

SupportAbstraction is depicted in Listing 7.6. It extends bw.base.cs.SubsetAbstraction<Marking, Transition, Place>. The listing omits only the constructor which passes its parameters to the super class.

SearchGuide which implements java.util.Comparator<Marking> in direct relation to the total order on markings chosen in Sect. 5.5.1 on p. 156.

PetriNet extends the abstract bw.base.Model<Marking, Transition>. It implements the method getAbstractionFor(...) (cf. Listing 7.7) which defines the hierarchy of the data structure to use by the (abstract) parent class. The search strategy SearchGuide is provided by the getSearchGuide() method and the computation of the pred-basis is implemented in the method getPredecessors(...) as described in Lemma 5.2 on p. 141.

Furthermore, pruning is implemented per static analysis (carried out in the method initializeBeforeModelChecking()) of the nets inequality P-invariants and accessed through the prune(...) method. Partial-order reduction is implemented in the method getBackwardEnabledTransitions(...). Since there is no branching in the pred-basis computation for Petri nets, the backward acceleration SSC is incorporated as described in Sect. 5.1.3 on p. 141.

We allowed our Petri net implementation to have a single initial state. It is returned by the method getInitialStates(). The finite basis of final states is accessible via getFinalStates(). Lastly,

the method `executeTrace(...)` is implemented to allow for forward execution of transition sequences.

`Parser` which implements `bw.interfaces.Parser<PetriNet>`. It reads a (textual) PEP [Gra97, Gra98] ll_net description[10] of a Petri net together with a description of the final states and returns a corresponding instance of the `PetriNet` class. In the configuration file `cfg/types.cfg` for our reference implementation, the line `.ll_-net=bw.impl.pn.Parser` expresses that an input file is to be read by this parser if its file name ends on `.ll_net`.

7.1.2.1. The Cost of Extending our Reference Implementation

Let us summarize the advantages of our framework: if a new system class already has a Java implementation, the main part of the additional effort to use our framework lies in finding the covering predecessor computation and a suitable well-quasi ordering. While it is very beneficial to have a search guide and indicator functions to employ our predefined data structures, it is not required. We believe that the cost to implement a backward reachability analysis around an existing implementation of a model outweighs the work needed to connect that implementation to our interfaces and abstract classes.

7.1.3. Unit Tests

While the description of the data structures of Ch. 6 on p. 160 is rather abstract, we had to exercise due diligence for their implementation. This embodied the creation of specialized structures like class `SortedUcs` for UCSs with a total order on states where the minimal state is accessible (like W in our framework Alg. 3.7 on p. 80), class `IterableUcs` for UCSs where elements are enumerable (like F in our framework), and class `Ucs` for UCSs where the states do not have to be accessed individually (like V in our framework).

For most of the classes we provide in the `bw.base` package and its subpackages, we have created unit tests using the JUnit framework to minimize the number of programming bugs in the reference implementation.

[10] PEP is available at `http://peptool.sourceforge.net/`. The format is described in `http://parsys.informatik.uni-oldenburg.de/~pep/Paper/formats_all.ps.gz`.

Specifically our implementations of the (abstract) generic data structures for UCSs that use the classes `EqualityAbstraction`, `TotalOrderAbstraction`, and `NoAbstraction`—with their interaction with the data structures used to incorporate a total order on states w.r.t. a search strategy—have been thoroughly tested.

We decided to employ two sets of test cases for our data structures: 1. hard-coded tests where we called the classes' methods in a meaningful order with sensible parameters and tested the return values and 2. "stress tests" that created tens of thousands of states that were added and removed repeatedly to the data structure in a randomized order. For both tests, we maintained a second, naive implementation of a respective data structure as a control to check the tested structure against.

Use of Java Assertions. Another means to test the framework's code as well as code provided by the user stems from our commitment to use Java assertions for the reference implementation.

Java assertions are statements in the form of `assert <condition> : <message>;` that are *ignored* by the Java virtual machine (JVM) unless instructed otherwise.[11] When disabled, they do not influence the performance, when enabled, the condition is checked and if it resolves to `false`, an `AssertionException` is thrown with the message as the exception's detail message.

We have added assertions to implement control tests of results returned both by our algorithms and by those the user has to implement. This way, the user may test many of the methods during experimental runs of his implementation without having to provide test code herself. For example, if a model inherits from our abstract class `bw.base.Model` and implements the `executeTrace(S state, Trace<T> trace)` method to allow for forward execution of a transition sequence (resulting in a collection of reached states) and the JVM is instructed to check assertions, then every computation of the covering predecessors is checked against the successor computation via `executeTrace(...)`.

[11] Start Java with argument `-enableassertions`, or `-ea`, on the command-line before naming the class to run, e.g. our tool with `java -ea bw.Runner examples/pn/hts.ll_net`.

7.1.4. Usage: An Example

To solve a coverability problem with the reference implementation BW of our framework, run the following command:

```
java bw.Runner input-file [options]
```

Currently, the supported file types are 1. `.lcs` for our textual representation of lossy channel systems, 2. `.ll_net` for Petri nets in the format used by the PEP tool, 3. `.spec` for Petri nets with transfer in the format used by the MIST2 tool, and 4. `.cfg` for our textual representation of context-free grammars.
The set of permitted command-line options depends on the type of the model under consideration and the employed analysis module—both may be defined by the user.[12]
As an example, we ran the program on the large case study of a holonic transportation system (HTS) [BR99, BR01] (cf. Sect. B.1 on p. 239 for a description) by executing the command

```
java bw.Runner examples/pn/hts.ll_net --trace
```

and got the output shown in Listing 7.9. Before the backward reachability analysis is conducted, the program performs some static analysis and information on the system under consideration is printed: 1. A number of invariants is computed (which are used for pruning). 2. Most oft the target states—i.e. the states in the basis of final states—are pruned (as they violate at least one of the found invariants). 3. There is one state in the basis of initial states. During the actual analysis, information on the current status is printed periodically.

Listing 7.9: Example output of the reference implementation

```
1 Computing inequality p-invariants...
2 Found 132 inequality p-invariants, covering 370 places.
3 12 state(s) in basis of target (37 pruned)
4 1 state(s) in basis of initial
5 10697 explored, 38049 created, 7.447s running time
6 Target states are coverable from state [1*p8, 1*p7, 1*p9] in 30
     step(s).
```

[12]See Sect. A.2 on p. 236 for command-line options of the Petri net implementation and the default analysis module.

```
7  That state is covered by initial state [1*p4, 1*p5, 1*p3, 1*p11,
       1*p8, 1*p10, 1*p7, 1*p1, 1*p9, 1*p6, 1*p2].
8  Trace: t38, t756, t497, t277, t256, t53, t330, t560, t409, t218,
       t518, t756, t38, t497, t277, t256, t53, t330, t560, (t38)^2,
       t330, t409, t44, t420, t174, t66, t330, t460, t253
9  Executing the trace from the found state leads to 1 state(s), the
       following 1 of which cover(s) some target state(s):
10 [1*p169, 1*p172, 1*p8, 1*p180, 1*p7, 2*p171] covers the state
       [1*p172, 1*p180] in the basis of the target.
```

The program's output ends with either stating that the final states are not coverable or by showing details on the covering transition sequence as in the case of our example. There, the state is shown which was reached by the backward analysis and the covering initial state—in this context, we call it `x`. In our example the final states (target states) are *bad* states. As the found state `x` contains far fewer tokens (on a smaller number of places) than the actual initial state, we observe a core of the problem with our Petri net where the unessential tokens of the initial state are omitted. With the well-quasi ordering for WSLTSs being a simulation relation, it is reasonable to examine minimal "problem states".

Lastly, the found transition sequence is printed that leads from the found state to the upward-closure of the final states. Since we used the `-trace` command-line option, the BW tool attempts to execute the found sequence of transitions beginning from an initial state and then verifies that the reached state—here, we call it `y`—indeed covers an element of the basis of final states. (In general, there may be several states reached via the transition sequence.) The state reached from the found state `x` via the found transition sequence may be smaller than the state reached from the covering initial states. Again, this may help to examine the problem of the bad states being reachable as the states `x` and `y` are potentially easier to understand than the actual system behaviour.

7.2. Experiments

In this section, we present and discuss the results of an extensive set of benchmarks we conducted to experimentally evaluate our algorithmic framework. Since we are neither aware of other tools for checking coverability in LCSs nor of a large enough set of benchmark models for a

meaningful experimentation, we present our experiments on LCSs and then turn the focus on Petri nets and Petri nets with transfer as the input models.[13]
After a short overview of the problem instances we use for benchmarking, we test and discuss

- how the different search space constructions (SSCs) and our simple search guidance influence the runtime and
- how a set of different tools for checking coverability stack up each other.

7.2.1. Lossy Channel Systems

For lossy channel systems, we implemented the following search space constructions: 1. Partial order reduction as described Abdulla et al. in [AKP97, AJKP98]. 2. Our backward acceleration (cf. Sect. 5.1 on p. 131). Additionally, the search is guided towards initial locations of the automata as described in Sect. 5.5.1 on p. 156. Our prototypical implementation was able to solve the coverability problem for the sliding window protocol with 30 messages in less than 3 minutes.
In the course of experimentation, a previously unknown bug in Schnoebelen's construction [Sch02] of a lossy channel system weakly computing the Ackermann function was discovered [SM12]:[14] In the definition of channel system T_n, which is supposed to transfer a number in unary notation from one channel to another, the number was accidentally increased by one by the very last transition to location x_n.[15]

7.2.2. PN and PNT Benchmark Models

In Table 7.1 properties of the considered 49 benchmark models are listed. The columns $|P|$ and $|T|$ denote the number of places and transitions in the Petri net (with transfer), whereas the column $|F|$ gives the number

[13] From the literature on LCSs, we know of the bounded retransmission protocol, the sliding window protocol (which subsumes the alternating bit protocol), and Schnoebelen's construction of the Ackermann function.

[14] We would like to thank Philippe Schnoebelen and Eike Möhlmann for discussions on this topic.

[15] The transition is labelled with actions $c_n?e, c_n!1e$, while they should read $c_n?e, c_n!e$.

of markings in the (not necessarily minimal) basis of the final states. The set of final states is not coverable for models where column "Cov." contains a cross. Numbers in column "Cov." are lengths of shortest known[16] transition sequence from an initial state to the upward-closure of the final states . Petri nets (with transfer) that are finite state systems (there is a bound on the number of tokens in the Petri net's marking) are marked with tick in the "Bound." column. Those models that are Petri nets with transfer (and not Petri nets) have a tick in column "Transf."
Most of the models come from the benchmark set of the MIST2 tool.[17] Those models with two numbers at the end come from the authors of the BFC tool.[18] The glotter model comes from M. Heizmann. Models set in *italics* are described in Appendix B on p. 239. We will discuss the tools in Sect. 7.2.4.
Note that the models vary heavily in the number of places and transitions: ranging from three places and two transitions up to over 700 places and nearly 800 transitions. Also, the maximal number of states in the basis of final states exceeds 63 000.

Table 7.1.: Properties of benchmark case studies

Model	$\|P\|$	$\|T\|$	$\|F\|$	**Cov.**	**Bound.**	**Transf.**
basicextransfer	3	2	1	×	✓	✓
basic-me	5	4	3	×	×	×
bingham-h250	253	501	63251	×	✓	×
consprod	18	14	1	×	×	✓
consprod2	18	14	1	×	×	✓
csm	14	13	1	×	×	×
csm-broad	13	8	1	×	×	✓
delegatebuffer	50	52	1	×	×	✓
delegatebuffer.15.1	50	52	1011	13	×	✓
delegatebuffer.16.1	50	52	6503	13	×	✓

Continued on next page

[16]As MIST2 performs a breadth-first search, we took the length of the transition sequences that it found as the shortest paths (if any). For the hts case study, MIST2 did not finish in time. We chose to present the length of the transition sequence returned by PETRUCHIO/BW in the table.
[17]`https://github.com/pierreganty/mist/tree/master/examples`
[18]We received the models in the context of personal communication.

Table 7.1 – continued from previous page

Model	$\|P\|$	$\|T\|$	$\|F\|$	**Cov.**	**Bound.**	**Transf.**
efm	6	5	1	×	×	×
examplelea	48	52	1	×	×	✓
ext-rw	24	22	1	×	×	×
ext-rw-smallconst	24	22	1	×	×	×
fms	22	20	1	×	×	×
fms2	22	20	26	×	×	×
german	12	8	1	×	×	✓
glotter	55	80	5	7	✓	×
hts	734	788	49	29	×	×
java	44	37	1	14	×	✓
java.10.0	44	37	551	10	×	✓
java.11.0	44	37	3256	10	×	✓
java2	44	38	1	×	×	✓
java2.10.2	44	38	595	10	×	✓
java2.11.2	44	38	3303	10	×	✓
kanban	16	16	1	48	×	×
km-nonterm.4.3	5	4	20	12	×	✓
km-nonterm.5.4	5	4	15	20	×	✓
km-nonterm.6.5	5	4	10	30	×	✓
lamport	11	9	1	×	✓	×
leabasicapproach	16	12	1	4	×	✓
leaconflictset	30	15	1	15	×	✓
lifo	7	10	1	×	×	✓
manufacturing	13	6	1	×	×	×
mesh2x2	32	32	1	×	×	×
mesh3x2	52	54	1	×	×	×
moesi	9	11	1	×	×	✓
moesi5	45	55	5	×	×	✓
multi-me	12	11	3	×	×	×
multipoll	18	21	1	×	×	×
newdekker	16	14	1	×	✓	×
newrtp	9	12	1	×	✓	×
peterson	14	12	1	×	✓	×
pncsacover	31	36	1	32	×	×
pncsasemiliv	31	36	1	10	×	×
queuedbusyflag	80	104	1	×	×	✓

Continued on next page

Table 7.1 – continued from previous page

Model	$\vert P\vert$	$\vert T\vert$	$\vert F\vert$	Cov.	Bound.	Transf.
read-write	13	9	1	×	✓	×
simplejavaexample	32	28	1	10	×	✓
transthesis	90	117	7	×	×	✓
				31 ×	41 ×	26 ✓

The benchmark models differ strongly in how challenging they are for the coverability checkers we test with. We will see that some problems that are hard to solve for one tool are readily solved by another and that the size of the model is no reliable indicator for the complexity of the problem.

Input Randomization. In the following sections, we run the benchmarks for each model repeatedly to compute decent median values for the runtime per model. As the order in which places, transitions, and arcs appear in the input file may influence random choices (that are non-deterministic on an abstract level of description)[19], we decided to "shuffle" every net for each benchmark iteration. By shuffling we mean to build the same net structure anew but in random order of object instantiation to alleviate the effect of the fixed ordering in the input file.

7.2.3. Experiments on SSCs and Search Guidance

To observe their effects on runtime, we have tested every combination of the methods described in Ch. 5 on p. 130. For each combination of backward acceleration (A), pruning by inequality P-invariants (I), partial order reduction (P), and search guidance (G), we let our tool try to solve the benchmark models. The ordering of the different SSCs is fixed to partial-order reduction (if used), then pruning (if used), then backward acceleration (if used). In order to get meaningful results in reasonable time, we limited the runtime per model. The schematic process of benchmarking the different combinations for each model is shown

[19] For Java in particular, the default hash value of an object depends on its place in memory which in turn depends on the state of the memory and the order in which objects are created.

in Alg. 7.1: for each attempt to solve a problem instance, the tool had 20 minutes and at least three attempts were started. When the first three attempts were finished in less than 15 minutes, the benchmark was repeated until the 15 minutes were reached. The idea behind this process is to get the average of a large number of benchmark iterations when a problem was solved quickly and to have at least three data points if the problem took longer to solve.

```
1 foreach opt ∈ 𝒫({ A, G, I, P }) do
2     i := 0;
3     reset clock x;
4     while i < 3 ∨ x < 15 minutes do
5         attempt to solve the model with opt and a timeout of 20
          minutes
6         i := i + 1
7     od
8 od
```

Algorithm 7.1: Benchmarking SSCs and search guidance.

The results of these benchmarks are shown in Fig. 7.2 (the benchmarks in total took 13 days and 18 hours, cf. Sect. C.2 on p. 269 for a tabularly listing of the results). It relates the number of problems that were solved within 20 minutes (per problem) to the median runtime it took to solve all those problems. For example, consider the line for the empty set of methods: 17 problems could be solved within 20 minutes each and it took roughly two seconds to solve those 17 problems. Using backward acceleration as the only method, also 17 problems were solved in 20 minutes each—but it took over a minute to solve them. On the right-hand side of the plot we indicated how many problems were solved by which combinations of methods where "GI±AP" stands for "search guidance and pruning together with any combination of backward acceleration and partial-order reduction", or, in our short form, GI, AGI, GIP, and AGIP.

Figure 7.2 allows us to rate the importance of search guidance and SSCs in the context of instantiating our framework for Petri nets (with transfer): First of all, the single most beneficial method is search guidance! With this technique alone, 45 problems are solved. Pruning was the next best method, as it enables our tool to solve all but one problem when

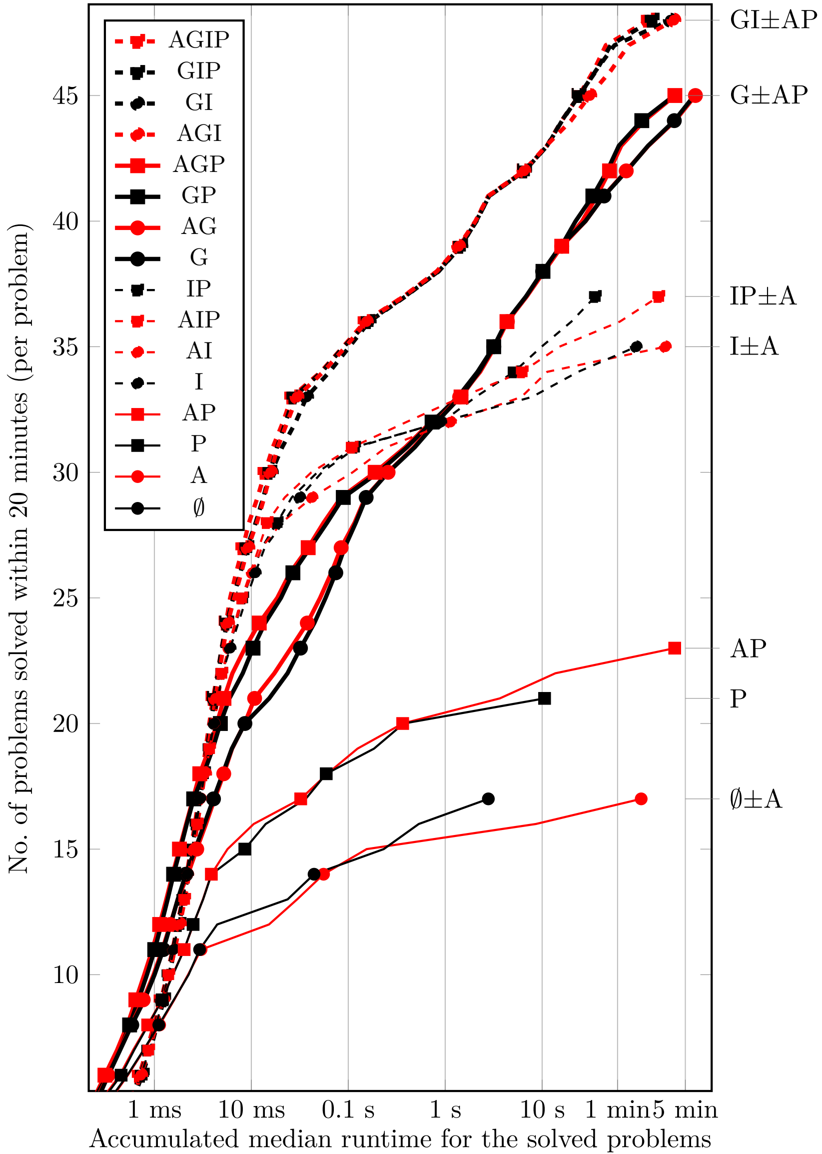


Figure 7.2.: Comparison of effects of SSCs and search guidance.

Model	GIP	AGIP	Factor
bingham-h250	**2.477**	2.577	0.961
delegatebuffer.15.1	**3.745**	3.758	0.997
delegatebuffer.16.1	12.345	**11.305**	1.092
examplelea	14.397	**10.092**	1.427
hts	63.682	**56.491**	1.127
java2	**4.671**	4.716	0.991
queuedbusyflag	8.113	**8.068**	1.006

Table 7.2.: Selected benchmark data for AGIP vs. GIP.

combined with search guidance. Then comes partial-order reduction and lastly our backward acceleration which appears to lower the performance of the tool as more calculations have to be carried out in order to find candidate sequences and acceleration candidates. Keep in mind that backward acceleration is a general SSC which is independent of the system class, whereas the aforementioned methods require a (possibly deep) understanding of the class to be effectively implemented. Moreover, the runtime seems to increase significantly only for small problems and if the search guidance is not active; in absolute terms, the effect is of little account as the increase does not exceed 100 milliseconds.

In conjunction with search guidance and the other methods backward acceleration tends to decrease the runtime for the more difficult problems as Table 7.2 shows. In that table, we listed the median runtimes in seconds for problems that took at least one second to solve with search guidance, pruning, and partial-order reduction (GIP) and compared it to the median runtimes when backward acceleration is also enabled (AGIP). Better runtimes are bold. We observe that the results differ only very slightly except for the harder problems. In summary, for the easy problems in our benchmark set, backward acceleration can result in a little slow-down (in Table 7.2 at most 4%); for the harder problems, it tends to result in a significant speed-up (of up to 42%).

7.2.4. Tool Comparison

For an experimental evaluation of our approach, we carried out an extensive comparison with the following tools.

MIST2 takes Petri nets (with transfer) as input and performs a breadth-first backward search with pruning (classical P-invariants are read from the input file) using the interval sharing-tree data structure [GMV$^+$07]. The tool contains several analysis algorithms [DRV01, GRV05, GRV06a, GRV08] but only the implementation of the backward reachability analysis supports Petri nets with transfer so that we consider only MIST2/BW. The tool runs a single thread.

`https://github.com/pierreganty/mist/`

BFC 2.0 is the current version of the tool of A. Kaiser et al. and is intended to solve coverability problems for *monotone dual-reference programs* which extend *thread transition systems* (TTSs) and are as expressive as Petri nets with transfer. The tool implements a variant of the *target set widening* algorithm [KKW12] and runs two threads: one for a forward analysis and one for a backward analysis. It also uses partial-order reduction (POR) techniques during the analysis.[20]

`http://www.cprover.org/bfc/`

BFC 1.0 is an older version of the BFC tool that performs better than BFC 2.0 on the type of systems we consider in our evaluation. As version 2.0, this tool uses a variant of the target set widening algorithm [KKW12] and runs two threads: one for a forward analysis and one for a backward analysis. However, it does not employ partial-order reduction.[21]

`http://www.cprover.org/bfc/`

PETRUCHIO/BW is our older coverability checker for Petri nets (with transfer) that we implemented for the tool PETRUCHIO[22] [SM12].

[20] During our benchmarks process, we spotted a spurious answer from this tool: it falsely reports the final states of the basicextransfer problem to be coverable. The bug has since been fixed but we were unable to rerun the benchmark for this tool due to time constraints.

[21] This older version contains bugs that lead to occasional deadlocks or segfaults during the benchmarks. In the runtime measurement of this tool, we have only considered attempts to solve problems that went without error (cf. Sect. C.3 on p. 286 for details).

[22] The PETRUCHIO tool [MS10] translates π-calculus processes into Petri nets w.r.t. the structural semantics [Mey09]. During the incremental construction of the corresponding Petri net, coverability problems have to be solved.

It was an early testing ground for some of the ideas presented in this thesis and employs pruning via inequality P-invariants, POR, backward acceleration and the search guidance of Sect. 5.5.1 on p. 156. The tool was developed over several years without having a clean architecture so that it is far from being easily extensible. We will see that on average this tool performs a bit better than the reference implementation of the framework. While the Java virtual machine may run multiple threads, e.g. for garbage collection, the analysis algorithm runs in a single thread.

`http://csd.informatik.uni-oldenburg.de/~critter/petruchio.tar.gz`

FRAMEWORK is the reference implementation of our algorithmic framework that we presented in the previous section. The implementation's focus lay on the creation of a framework with multiple layers of abstraction that allows for straightforward instantiation as opposed to maximal performance. While the Java virtual machine may run multiple threads, e.g. for garbage collection, the analysis algorithm runs in a single thread.

`http://csd.informatik.uni-oldenburg.de/~critter/bw.tar.gz`

Overall Benchmark Results. We divide the results of benchmarks of the five tools into three figures that we discuss collectively. Figure 7.3 shows the accumulated median runtimes for the number of solved problems. For example, the reference implementation of our framework took ca. 1 millisecond to solve its 10 “fastest” problems and it solved all 49 problem instances in ca. 5 hours total. The MIST2 tool solved only 40 problems and took roughly 2 days for all of them. In this figure the runtimes are median values of a number of repeated benchmarks per problem. The graph leads us to the conclusion that while PETRUCHIO/BW is the fastest of the five tools, despite the focus on extensibility, our reference implementation is able to keep up with it performance-wise. The MIST2 tool is able to solve more problems than the BFC tool and the older version of BFC is more suited for the models we consider in our benchmarks.

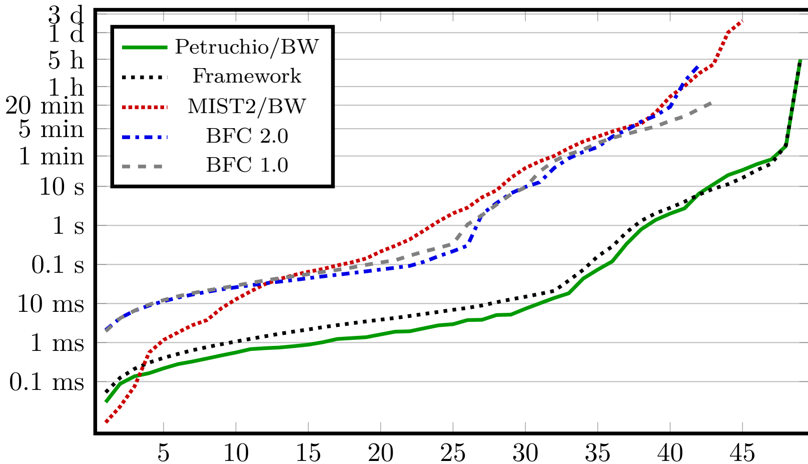


Figure 7.3.: Tool comparison: Median time.

The other two figures, Fig. 7.4 and Fig. 7.5, show maximum runtimes and the upper bound of the 97%-confidence interval—meaning that in 97% of the benchmark repetitions, the presented runtime was not exceeded. We observe that the overall picture looks similar to the median runtimes shown in Fig. 7.3. However, the tool Petruchio/BW experiences (maximal) outliers that are small in absolute value but large in comparison to the other four tools. The figures show that our reference implementation is superior even to Petruchio/BW when extremal runtimes are taken into consideration.

Set-Up. The benchmarks were run on an Intel Xeon E5430 machine with eight 2.66 GHz CPUs each with 6144 KB cache, which shared 32 GB of memory running Gentoo Linux 2.6.34-gentoo-r1 and Java SE Runtime Environment build 1.6.0_24-b07 (Java HotSpot 64-Bit Server VM build 19.1-b02, mixed mode). During measurement, no two benchmarks were run at the same time, leaving most CPUs spare for background processes. For each case study, the benchmarks were repeated at least three times and for at least two hours before computing the median runtime. No timeout was imposed on each iteration of an attempt to solve the

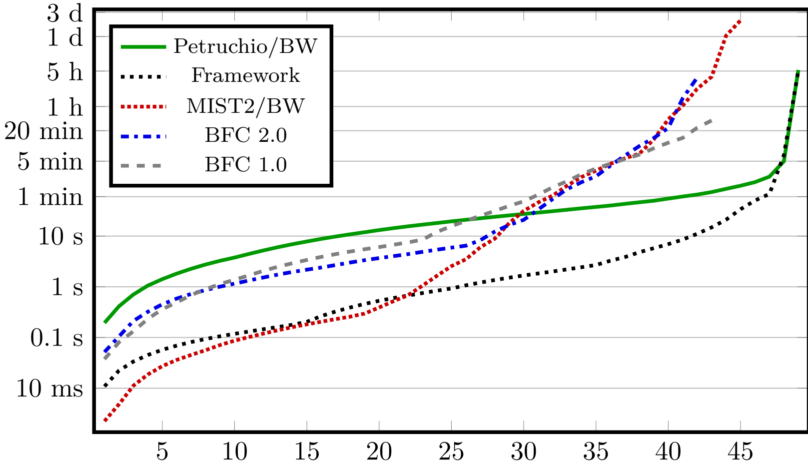


Figure 7.4.: Tool comparison: Maximum time.

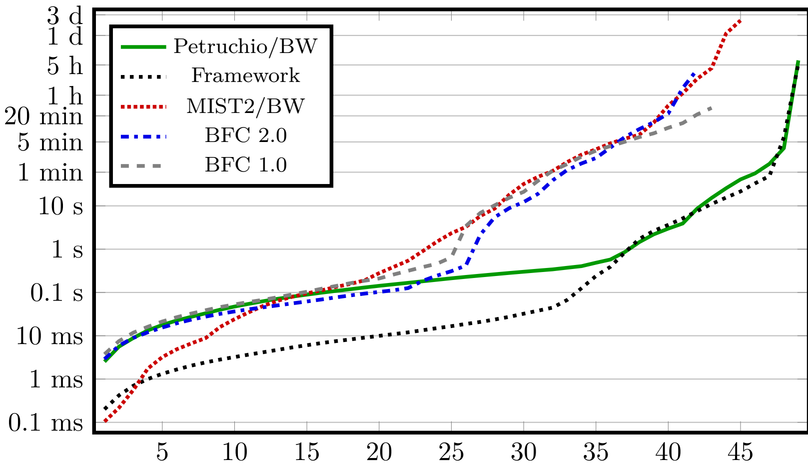


Figure 7.5.: Tool comparison: 97%-confidence interval.

given problem. Due to the nature of Java's Virtual Machine, which has a "slow" start-up phase, the standard deviation for benchmarks, for which the reference implementation and PETRUCHIO/BW were capable to answer in well under one second, significantly decreases (mostly to values near 0.2 milliseconds), when more iterations of the benchmarks are run (however, the average runtime stays virtually the same). To get a fair comparison, we refrained from countering this circumstance by running the benchmarks of our Java programs over a longer period of time. To extend the runtime of the benchmarks for all five tools was impractical.

Detailed Results. More detailed results of the comparison are depicted in Fig. 7.6 and Fig. 7.7, where runtimes are given on a logarithmic scale. The models are ordered according to the median runtime of our tool. For models where final states are coverable, the median length of the found trace is printed atop the bar for the tool's runtime. Interestingly, our method tends to return longer witness paths than the other tools do. This is a result of guided search and the use of our acceleration SSC.

While our tool was able to present a transition sequence of length 29 for the holonic transportation system (HTS)[23]—a benchmark case study described in [MKS09]—in 76 seconds, MIST2 was stopped after running for nine weeks without results. However, for that particular problem the BFC 2.0 tool was even faster: it solved the HTS problem in a little over 3 seconds. The problem instance pncsacover [Fin93], a PN modelling a portion of an authorization protocol, was solved in roughly 18 milliseconds median runtime over ca. 295 000 iterations by our reference implementation and in roughly 47 seconds median runtime by MIST2 over 152 iterations.

Three of the benchmark models contain constellations of PNT arcs[24] which are not supported by MIST2 and the program SPEC2TTS that creates the input files for the BFC tool. For the basicextransfer model, BFC 2.0 outputs a spurious transition sequence. In four cases, the BFC tool attempted to allocate more than 32 GB of memory and was stopped. In total, the benchmarks of the five tools took 28 days and 8 hours to

[23] `http://csd.informatik.uni-oldenburg.de/~critter/pet-bw-examples.tar.gz`

[24] A place p is reset by transition t (input arc of t) which also adds a fixed number of tokens to p (output arc of t). e.g. a net with $W(p,t) = p$ and $W(t,p) = 5$.

complete—not counting the roughly 63 days MIST2 ran without completing the HTS problem. See Sect. C.3 on p. 286 for numerical benchmark data.

Related Work. While we focused on the system classes of Petri nets and Petri nets with transfer in our benchmarks and economized on only 49 problem instances, of which the majority comes from a competitor tool, there exists a plethora of benchmark case studies and tools.

Most of the benchmark case studies we are aware of (e.g. cf. [Cor96, Pel06, Pel07]) are designed with explicit model checking, state space generation, deadlock checking, and reachability problems in mind. Although coverability checkers can be used to semi-decide some reachability problems—as the uncoverability of a set of final states implies their unreachability but not vice versa—, we find it advisable to employ well-established and efficient reachability checkers for these problems.

There is a large set of more specialized coverability checkers, some of which we discuss in the following. However, we cease to include them in our benchmarks as they are tailored for specific problems. In [MMW13], a SPIN-based coverability checker for provenance tracking in concurrent programs is introduced which is several orders of magnitude faster for their benchmark set of problems than the Petri net coverability checkers MIST2 and Petruchio/BW. An incremental, inductive procedure to check coverability of a subclass of WSTSs is given in [KMNP13] where the authors show that their implementation is several orders of magnitude faster on a set of Petri net benchmarks than the backward analysis MIST2/BW, the expand, enlarge, and check (EEC) procedure in MIST2/EEC [GRV06b, GRV05], and the target set widening technique of BFC.

Earlier work compared the performance of coverability checkers against the polyhedron-based tool HyTech of Henzinger et al. [HHWT97] which is intended for the model checking of hybrid systems. In [DRV01, DRV02], Delzanno et al. evaluate their coverability checker based on covering sharing trees (CSTs) and find a vast performance improvement in comparison to HyTech. Bingham et al. produced a tool BUCSUB [Bin05, BH05] based on binary decision diagrams (BDDs) for a subclass of WSTSs and show that it outperforms both the CST-based approach of Delzanno et al. and HyTech.

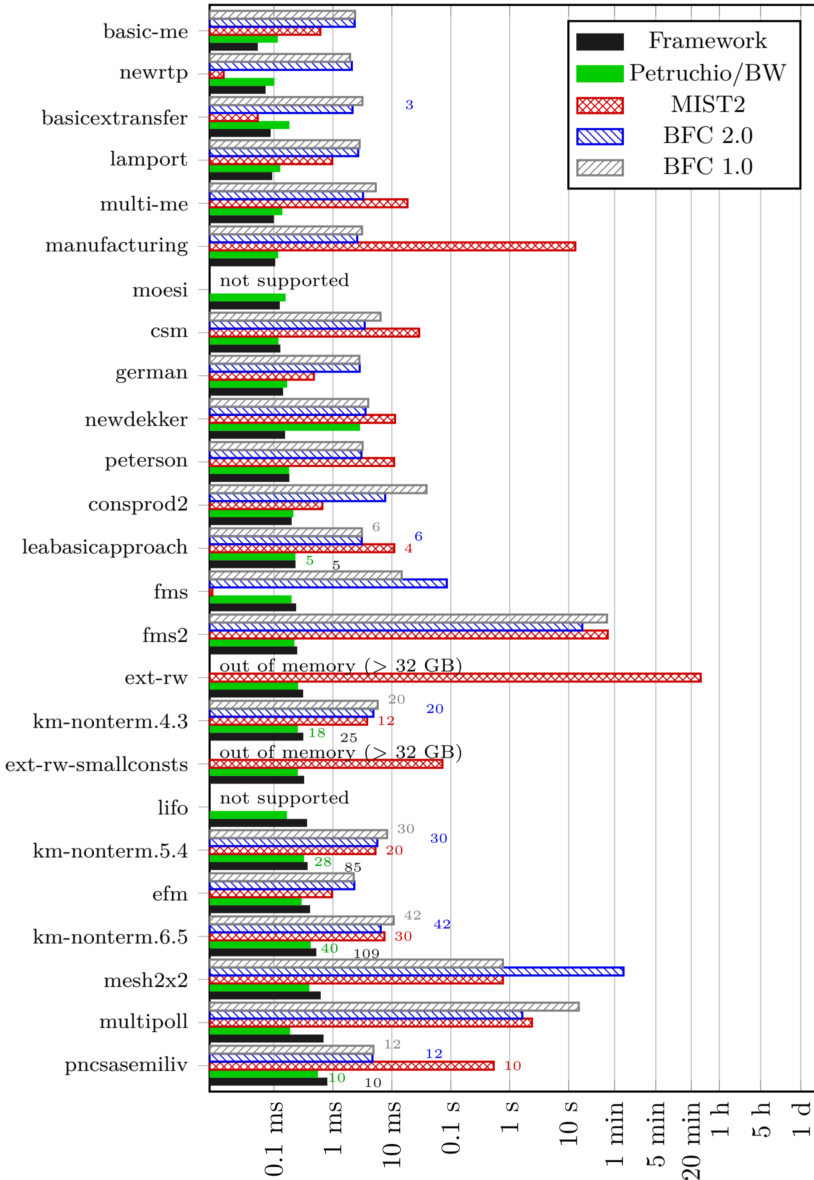


Figure 7.6.: Detailed benchmark results I.

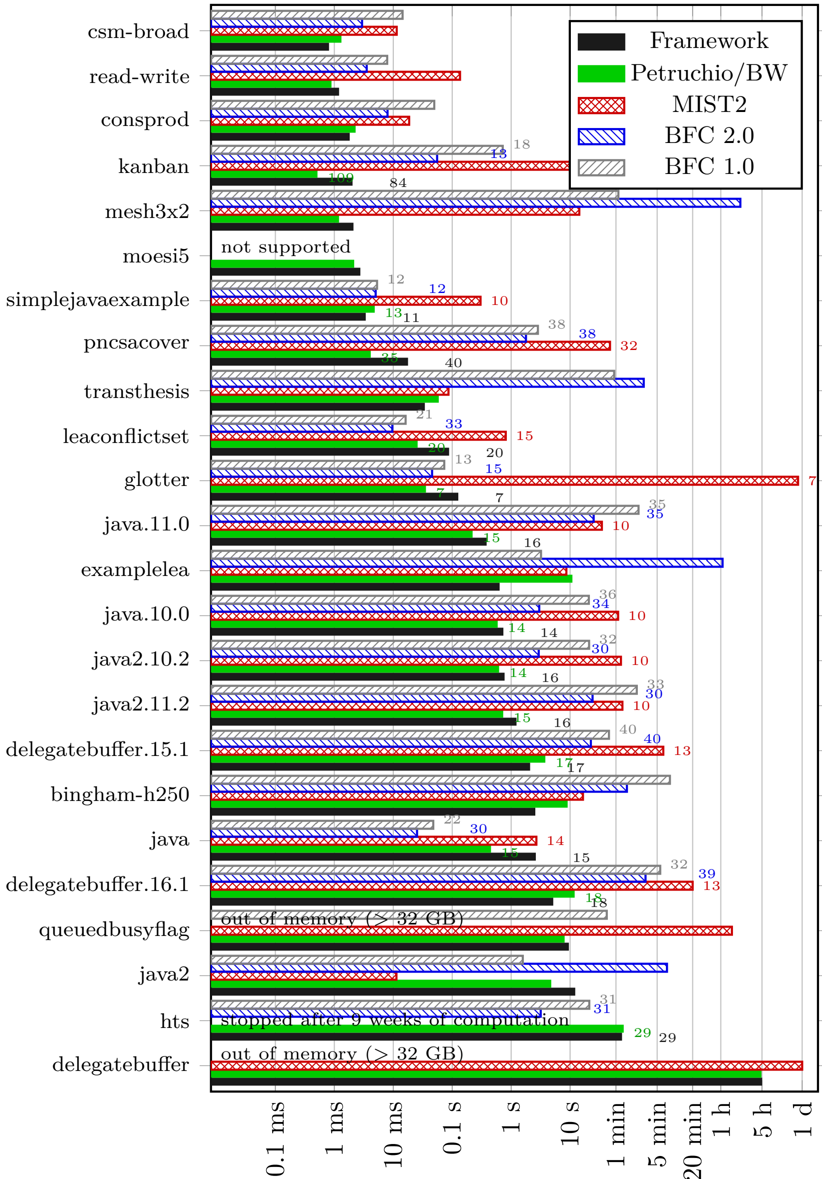


Figure 7.7.: Detailed benchmark results II.

Concluding Remarks. In our opinion, while the five tools we tested all perform well on our set of PN and PNT benchmarks, our reference implementation provides excellent performance in terms of both the observed median runtime and the observed worst-case runtime even in comparison to state-of-the-art competitor tools.
In the discussion of related work we have seen that fine-grained observations and optimizations can be exploited and employed for more specific system classes to gain a significant speed-up. However, our intention is to not to compete with each of the above tools but to provide a framework that is an extensible platform for experimentation and a test bench for novel ideas.
Our framework and its reference implementation provide adequate means for *rapid prototyping* when investigating new approaches in terms of systems under consideration, efficient data structures, and optimizations such as search-space constructions and search guidance.

CHAPTER EIGHT

Conclusion

The Guide is definitive. Reality is frequently inaccurate.
— Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Contents

8.1 Summary 225
8.2 Directions for Future Work 227
8.2.1 Algorithmic Framework and Distance 227
8.2.2 Implementation 229
8.2.3 Well-Structured Transition Systems 230

In this chapter, we summarize the results of this thesis and address directions for future work.

8.1. Summary

This thesis is divided into two parts: *an algorithmic framework* and *instantiating the framework*.

In the first part of this thesis, we have developed an algorithmic framework for checking coverability of well-structured labelled transition systems (WSLTSs) that generalizes the established basic backward coverability analysis (BR). We gave a novel, axiomatic proof of total correctness of the BR that allows for a precise understanding of the algorithm. Subsequently, we created incremental extensions of the basic analysis and arrived at our parametrized framework algorithm. An abstract distance and a witness function were used to formalize the requirements for adequate instantiations, called search space constructions (SSCs), of the framework in a concise fashion. We discussed high-level arguments for partial correctness and termination of the framework and conducted an axiomatic proof of total correctness.
The thesis' second part was concerned with an intermediate concretization of our algorithmic framework that reduces the costs to instantiate it for specific models. We began by the introduction of a novel search space construction, called *backward acceleration*, which is applicable for the whole class of WSLTSs. We showed that it can reduce the search space by identifying and cutting out recurring patterns in the explored transition sequences. We studied how it is to be implemented first in general and second, more efficiently, for the case of Petri net models. Thereafter, we showed that the notion of SSCs is conservative in the sense that it captures the well-known optimizations pruning—for which we extended the common technique for Petri nets by inequality P-invariants—and partial-order reduction. Furthermore, we examined methods to combine different search space constructions and illustrated the effects. We discussed means to incorporate search guidance into our framework and implemented it on the example of syntactic distance (and weight) for Petri nets and lossy channel systems (LCSs).
In order to allow for the virtually effortless instantiation of our framework, we presented the simple concept of employing necessary conditions to partition finite bases of upward-closed sets of states. These partitions were used to reduce the number of comparisons that have to be made during a coverability analysis. We identified basic types of necessary conditions and discussed how two of them can be implemented using well-understood data structures. For a third type of condition, we introduced a novel data structure, called *powerset search trees*, and showed their beneficial effect on the runtime in an preliminary experiment. We discussed how data structures induced by necessary conditions can be

structured into hierarchies. We presented possible instantiations for LCS configurations and experimented with data structures for sets of Petri net markings.
We implemented our abstract framework, tested it thoroughly, and showed its practicality by illustrating how we instantiated it for Petri nets. In an empirical evaluation of the different search space constructions and the search guidance, we found that a (simple) search guidance is the most effective means to improve the programs performance. We discussed the positive results for our model-independent SSC backward acceleration and documented its impact on the tools runtime. We carried out a comprehensive set of benchmark experiments in order to draw comparisons between five coverability checkers for Petri nets with transfer. The data showed excellent results for our algorithmic framework and lead us to the conclusion that it is well suited as a test bed for newly found optimizations and for prototyping coverability checkers for novel classes of WSLTSs.

8.2. Directions for Future Work

We classify the starting points for future work into three groups: 1. those in context of our algorithmic framework, 2. those concerned with the reference implementation, and 3. those on the general topic of well-structured transition systems (WSTSs).

8.2.1. Algorithmic Framework and Distance

For the development of our algorithmic framework, we made some design decisions in the extensions of the basic backward analysis and introduced abstract constraints for adequate instantiations of our framework.

Memory Footprint. In the framework we presented in Sect. 3.3 on p. 79, the finite bases V and W are disjoint. The algorithm might be altered to have W being a subset of V.[1] This could be accomplished by storing only the set V and *marking* those states that are in W as schematically shown in Alg. 8.1.

[1] We would like to thank Jochen Hoenicke for the discussion on this topic.

```
1  V := minimize(F);
2  foreach y ∈ V do
3      mark y
4  od;
5  while there is a marked state in V do
6      x := select_marked(V);
7      unmark x;
8      if x ∈ ↓I then
9          V := { x } ;
10     else
11         foreach y ∈ opb(x) do
12             if y ∉ ↑V then
13                 mark y;
14                 V := minimize(V ∪ { y } )
15             fi
16         od
17     fi
18 od
```

Algorithm 8.1: Suggested algorithm for a reduced memory footprint.

The benefit of this approach is that there is only a single basis of an upward-closed set to be stored. This reduces the memory consumption and may increase the algorithm's performance as the minimal basis property has to be maintained for a single set only. Additionally, non-minimal states that are removed from V are automatically omitted in the search as they cannot be selected.

Distance Function. An anonymous reviewer of our paper [SM12] posed an interesting question: Does our approach that requires search space constructions to be *distance reducing* (cf. Sect. 3.4.1 on p. 82) extend to other analysis methods that are based on fixpoint iterations for WSTSs? The reviewer suggested to investigate if the notion of distance reduction is applicable to the results of Baier et al. [BBS06] who prove a general finite convergence property for "upward-/downward-guarded" fixpoint expressions over well-quasi-ordered sets. They show that theit fixpoint expressions—which allow for stating richer temporal properties than just

coverability—can be evaluated symbolically by an iterative procedure in a finite number of operations.

8.2.2. Implementation

Due to the use of interfaces, most instantiations of data structures and algorithms for the reference implementation of our framework can be exchanged easily. Therefore, we can understand the implementation as being wide open to change. There are several topics offering themselves as natural starting points for future work.

Application. The self-evident next step is to take our reference implementation and create prototype coverability checkers for classes of well-structured transition systems. De Boer et al. were able to show that restricted actor systems are well-structured transition systems [BJLZ12]. It is desirable to instantiate our framework for the class of restricted actor systems.

Data Structures. In Ch. 6 on p. 160 we have discussed how generic data structures for finite sets of states can be implemented. It would be interesting to see how they compare directly to the data structures of competing approaches. Covering sharing trees, interval sharing trees, binary decision diagrams, and multi-valued decision diagrams have been used for the coverability analysis of Petri nets or even larger classes of well-structured transition systems (cf. Sect. 6.1 on p. 160). If they were implemented in our instantiation of the reference implementation for Petri nets, an immediate empirical comparison could be commenced.

Parallel Processing for Data Structures. As addressed in Sect. 6.7 on p. 185, the data structures induced by necessary conditions are used to partition finite sets of states and therefore produce disjoint subsets. Read operations and some write operations could easily be carried out in parallel on disjoint subsets. The data structures for total order conditions and subset conditions may be extended to allow for searching through several subsets at the same time. The method might even be extended to enable the parallel execution of several different operations accessing the data structure for a single finite set, i.e. checking if x is in $\uparrow V$ in

parallel to checking if y is in $\uparrow V$. The correctness of such methods has to be shown and it has to be checked if they result in an actual performance gain.

8.2.3. Well-Structured Transition Systems

We have two suggestions for future work on well-structured transition systems that are not necessarily tied to our algorithmic framework.

Subclasses of Well-Structured Transition Systems. In Sect. 1.2 on p. 6 we referenced several related approaches that consider only subclasses of WSTSs. Since Finkel and Schnoebelen identified several abstract classes of WSTSs [FS01], Bingham and Hu introduced "nicely sliceable" WSTSs [BH05], Chambart et al. used "trace bounded" WSTSs [CFS11], and Kloos et al. [KMNP13] analysed "downward-finite" WSTSs. How are these classes connected to those defined by Finkel and Schnoebelen? As some of these classes cover Petri nets and lossy channel systems, is there a *common* property that our framework can easily exploit to increase its performance?

An Easily Comprehensible Example of a WSLTS. Well-structured transition systems are rather abstract. In conversations outside of the scientific community we experienced difficulties explaining the subject when concepts like Petri nets or lossy channel systems were unavailable. We believe that it is meaningful to create a WSLTS or an analogy that is comprehensible for people that are not considered insiders.
As an initial idea, we would imagine a card[2] game that meets the following requirements.

1. There is an infinite number of states—perhaps by introducing an infinite pack of cards or used cards being returned to the pack.

2. The notion of a state is clear—perhaps in the style of LCSs with a control state (a turn indicator) and a collection of channel states (the player's hands).

[2] Any type of game where players may gather an unbounded amount of resources could work.

3. There is a well-quasi ordering on states that is a simulation w.r.t. the rules of the game—perhaps players may choose a card to use from their hands; having more cards increases the options to choose from.

4. Transitions are labelled—the name of a card and the name of the player using that card may suffice.

5. There is a winning move or an upward-closed set of winning states.

6. The stakes are high; it is tangible why it is important not to lose—this motivates the need for model checking.[3]

7. It is not (or hardly) a game of chance, i.e. the outcome should not depend on some randomized order of cards in the pack.

8. It is be easily agreeable that there is a large number of likewise card games that all are WSLTSs.

Of course, some requirements might be relaxed if it serves the tangibility of the game.

[3]The loser(s) might be obliged to do the dishes.

PART III

Appendices

APPENDIX A

Implementation Details

Contents

A.1 Generics 235
A.2 Command-Line Options 236

In this appendix, we comment on some conceptual details of the reference implementation.

A.1. Generics

In the example of Listing 7.1 on p. 194, we see that the head of the interface of a simple state (one without a sequence of transitions) reads `SimpleState<S extends SimpleState<S>>`, meaning that it depends on the type parameter `S`. The type parameter `S` has a constraint attached: It is required to extend the type `SimpleState<S>`. While this constraint is circular and thus may be confusing, it is a standard technique and enables us to define the comparison operation to be between two states of the same type.
In Listing A.1, we give two other approaches to exemplify the use of generics. If we had not used generics as in interface `SimpleStateOne`, we

would have to specify the type of parameter `s` of the comparison operations explicitly to be `SimpleStateOne` which forces any implementation of a state to be comparable to any arbitrary implementations of a state, e.g. Petri net markings and lossy channel system (LCS) configurations would have to be comparable. In case of a simple generic type parameter as in interface `SimpleStateTwo`, we are not allowed to call the comparison operation on an abstract level, i.e. `SimpleState<S> state = ...; if(state.isLessThan(state)) ...`, as the type parameter and the interface are not related.
With a type parameter as defined in the actual interface `SimpleState`, we are able to implement, e.g, PN markings and use them even on an abstract level without effort: The `SimpleMarking` class implements the `SimpleState` interface with itself as the type parameter, resulting in objects of that class being comparable to other objects of that particular class—as forced by the interface.

Listing A.1: Choice of generic types for states

```
1  public interface SimpleStateOne {
2    boolean isLessThan(SimpleStateOne s);
3    // ...
4  }

6  public interface SimpleStateTwo<S> {
7    boolean isLessThan(S s);
8    // ...
9  }

11 public class SimpleMarking implements SimpleState<Marking> {
12   public boolean isLessThan(Marking s) { ... }
13   // ...
14 }
```

A.2. Command-Line Options

In Listing A.2 we have listed the (abbreviated) options for the framework's reference implementation BW with the Petri net implementation and the reference implementation `bw.analysis.Analysis` of our framework algorithm.

Listing A.2: Command-line options of the reference implementation

```
Usage:
  java bw.Runner input-file [options]
Options:
--help              prints this help
--help-more         prints help of loaded model
                    / analysis module
--quiet             short for "--log-level␣OFF"
--trace             executes trace
--no-acceleration   disables acceleration
--shortest          searches for shortest trace
--shuffle           model = model.shuffle before analysis
--parser class      uses given parser to load input
--analysis class    uses given class for analysis
--log-level level   sets log level to use
--properties file   sets file to read for the configuration

Options of bw.impl.pn.PetriNet:
--search-order order sets search order to use.
                    one of the following:
  depth-first       Explore states with longest
                    traces first; if equal let the greedy
                    search guide decide.
  breadth-first     Explore states with shortest
                    traces first; if equal let the greedy
                    search guide decide.
  greedy            Use the greedy search guide.
  a-star            Explore states with minimal
                      trace length + distance heuristic
                    first; if equal let the greedy search
                    guide decide.
--no-partial-order  disables partial order reduction
--no-pruning        disables pruning

Options of bw.analysis.Analysis:
--print-search-space file writes a GraphViz DOT
                          representation of the explored
                          search space
```

APPENDIX B

Sources of Selected Case Studies

Contents

B.1 Holonic Transportation System 239
B.2 PNCSA Protocol 246
B.3 Kanban Production System 250
B.4 Delegate Buffer Program 255

This appendix describes several more complex case studies together with associated control-state reachability problems and gives source files as inputs to a model checker.

B.1. Holonic Transportation System

The holonic transportation system from [BR99, BR01] is an industrial case study from the automated manufacturing domain consisting of autonomous transport vehicles carrying work pieces between machine tools, and input/output store.

The case study can be understood as a workshop in which e.g. drilling, sawing and bending machines are located and metal plates have to be drilled, then sawed and finally bent, whereas metal tubes are sawn first, then drilled and lastly bent.
Work pieces carry information on the order in which they have to be processed by machine tools. Whenever a work piece arrives at the input store or exits a machine tool (by being placed on the respective output buffer), a message is sent to the transport vehicles, upon which an idle vehicle fetches it and delivers it to the machine tool or store next in order of the work piece. Analogously, machine tools and stores send a message, when their respective input buffer is free for another work piece.
In a refined model, human mechanics are introduced, machine tools and transport vehicles may fail. Also, these refined, multi-functional vehicles may act as rescue vehicles and are called robots. In failure mode, a robot sends a message to an idle robot stating its point of failure to allow the rescue vehicle to perform the remaining part of the transportation. After the work piece has been taken away from the broken robot, a mechanic is called who may repair the robot. Machine tools may fail only when a work piece is being processed and they too send a message to the mechanic who may repair it.
The case study was formalized in [MORW04] and adapted for the π-calculus in [Gri07] and model checked using Petri net translations from [Mey08]. In [MKS09] a deadlock of Gringel's implementation was found using net unfoldings and a correction is proposed.
The deadlock-free, refined version of [MKS09] is given in Listing B.1, modelling three different types of machine tools with two instances each, an unbounded number of robots and work pieces, one input buffer, one output buffer, and one mechanic.

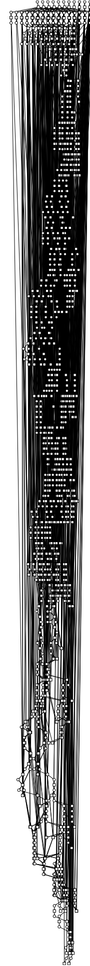


Figure B.1.: Overview of a Petri net representation of the HTS

An (unbounded) Petri net representation has been automatically generated by the tool Petruchio [MS10], consisting of 734

places, 788 transitions and 2443 arcs. As Fig. B.1 indicates, the net easily exceeds the size for which manual model checking appears to be practical.

The property to be checked represents a situation where there are at least two robots in rescuing mode, one in stage A and one in stage B (i.e. processes where either process identificator `RHTV_A` or `RHTV_B` can perform a transition), which both fail at the same time and call for the mechanic. Due to the two different paths a work piece can take through the process, there are seven Petri net places for each of the malfunctioning rescue robots. The property consists of 49 individual control-states each of which is reachable. The shortest known trace, leading to a covering control-state, consists of 29 steps in the generated Petri net.

For the files `hts.pi.ll_net` and `hts.pi.spec` attached to the digital version of this document the places corresponding to critical control-states are p172, p358, p383, p528, p588, p593, p696 for `RHTV_A` and p180, p193, p375, p465, p503, p612, p617 for `RHTV_B` while the trace to a state which marks p172 and p193 is t756, t756, t38, t497, t277, t256, t238, t330, t140, t409, t218, t114, t38, t497, t277, t256, t330, t406, t38, t38, t409, t44, t330, t420, t145, t768, t601, t330, t617.

Listing B.1: π-Calculus process of the holonic transportation system ([MKS09])

```
1  main
2  SYSTEM :=
3    ( MECHANIC(mtBroken, htvBroken)
4    | IN(createChan, id_in, htvC)
5    | OUT(out, id_out)
6    | MT(id_mt_1, mt_1, htvC, mtBroken,
7      i_1, Iyes_1, Ino_1, Iempty_1, Ifull_1, Iinput_1,
           Ioutput_1,
8      o_1, Oyes_1, Ono_1, Oempty_1, Ofull_1, Oinput_1,
           Ooutput_1)
9    | MT(id_mt_2, mt_2, htvC, mtBroken,
10     i_2, Iyes_2, Ino_2, Iempty_2, Ifull_2, Iinput_2,
           Ioutput_2,
11     o_2, Oyes_2, Ono_2, Oempty_2, Ofull_2, Oinput_2,
           Ooutput_2)
12   | MT(id_mt_3, mt_3, htvC, mtBroken,
13     i_3, Iyes_3, Ino_3, Iempty_3, Ifull_3, Iinput_3,
           Ioutput_3,
14     o_3, Oyes_3, Ono_3, Oempty_3, Ofull_3, Oinput_3,
           Ooutput_3)
15   | MT(id_mt_4, mt_1, htvC, mtBroken,
```

```
16     i_4, Iyes_4, Ino_4, Iempty_4, Ifull_4, Iinput_4,
          Ioutput_4,
17     o_4, Oyes_4, Ono_4, Oempty_4, Ofull_4, Oinput_4,
          Ooutput_4)
18   | MT(id_mt_5, mt_2, htvC, mtBroken,
19     i_5, Iyes_5, Ino_5, Iempty_5, Ifull_5, Iinput_5,
          Ioutput_5,
20     o_5, Oyes_5, Ono_5, Oempty_5, Ofull_5, Oinput_5,
          Ooutput_5)
21   | MT(id_mt_6, mt_3, htvC, mtBroken,
22     i_6, Iyes_6, Ino_6, Iempty_6, Ifull_6, Iinput_6,
          Ioutput_6,
23     o_6, Oyes_6, Ono_6, Oempty_6, Ofull_6, Oinput_6,
          Ooutput_6)
24   | ROBOTS(htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C)
25   | WPs(createChan, mt_1, mt_2, mt_3, out)
26   );

28 ROBOTS(htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C) :=
29   ( ROBOTS(htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C)
30   | new r.ROBOT(r, htvC, htvBroken, Ichan_A, Ichan_B,
        Ichan_C)
31   );

33 WPs(createChan, mt_1, mt_2, mt_3, out) :=
34   ( WPs(createChan, mt_1, mt_2, mt_3, out)
35   | new wp.CreateWP(wp, createChan, mt_1, mt_2, mt_3, out)
36   );

38 // Create a work piece (two different processing orders)
39 CreateWP(wpID, createChan, addressMT_1, addressMT_2,
      addressMT_3, addressOUT) :=
40   createChan<wpID>.
41   ( WP(createChan, wpID, addressMT_1, addressMT_2,
        addressMT_3, addressOUT)
42   + WP(createChan, wpID, addressMT_2, addressMT_1,
        addressMT_3, addressOUT)
43   );


46 // A work piece (unrolled the definition)
47 WP(createChan, id, addr_1, addr_2, addr_3, addr_4) :=
48   id<addr_1>.id<addr_2>.id<addr_3>.id<addr_4>.id().
49   WP_OUT(createChan, id, addr_1, addr_2, addr_3, addr_4);

51 // Spawn new work piece when one has been completely
      processed
```

```
52 WP_OUT(createChan, id, addr_1, addr_2, addr_3, addr_4) :=
53   createChan<id>.WP(createChan, id, addr_1, addr_2, addr_3,
       addr_4);

55 // The input store
56 IN(createChan, storeID, htvC) :=
57   createChan(wp).htvC<storeID>.storeID<wp>.
58   IN(createChan, storeID, htvC);

60 // The output store
61 OUT(address, storeID) :=
62   address<storeID>.storeID(wp).wp<>.OUT(address, storeID);

64 // A multi functional robot
65 ROBOT(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C)
     :=
66   ( htvC(buffer).HTV(htvID, htvC, htvBroken, Ichan_A,
       Ichan_B, Ichan_C, buffer)
67   + Ichan_A(buffer).RHTV_A(htvID, htvC, htvBroken, Ichan_A,
       Ichan_B, Ichan_C, buffer)
68   + Ichan_B(wp, dest).RHTV_B(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C, wp, dest)
69   + Ichan_C(wp, buffer).RHTV_C(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C, wp, buffer)
70   );

72 // A holonic transport vehicle with interrupts
73 HTV(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C,
     buffer) :=
74   ( Interrupt_A(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
       Ichan_C, buffer)
75   + tau.buffer(wp).wp(dest).HTV_B(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C, wp, dest)
76   );

78 RHTV_A(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C,
     buffer) :=
79   tau.buffer(wp).wp(dest).dest(buffer).tau.buffer<wp>.
80   ROBOT(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C);

82 Interrupt_A(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
     Ichan_C, buffer) :=
83   ( Ichan_A<buffer>.HTV_defect(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C)
84   + htvBroken<htvID>.htvID().RHTV_A(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C, buffer)
85   );
```

```
86 HTV_B(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C,
       wp, dest) :=
87  ( Interrupt_B(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
       Ichan_C, wp, dest)
88  + dest(buffer).HTV_C(htvID, htvC, htvBroken, Ichan_A,
       Ichan_B, Ichan_C, wp, buffer)
89  );

91 RHTV_B(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C,
       wp, dest) :=
92  tau.dest(buffer).buffer<wp>.
93  ROBOT(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C);

95 Interrupt_B(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
       Ichan_C, wp, dest) :=
96  ( Ichan_B<wp, dest>.HTV_defect(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C)
97  + htvBroken<htvID>.htvID().RHTV_B(htvID, htvC, htvBroken,
        Ichan_A, Ichan_B, Ichan_C, wp, dest)
98  );

100 HTV_C(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C,
       wp, buffer) :=
101  ( Interrupt_C(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
       Ichan_C, wp, buffer)
102  + tau.buffer<wp>.
103    ROBOT(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C
          )
104  );

106 RHTV_C(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C,
       wp, buffer) :=
107  tau.buffer<wp>.
108  ROBOT(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C);

110 Interrupt_C(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
       Ichan_C, wp, buffer) :=
111  ( Ichan_C<wp, buffer>.HTV_defect(htvID, htvC, htvBroken,
       Ichan_A, Ichan_B, Ichan_C)
112  + htvBroken<htvID>.htvID().RHTV_C(htvID, htvC, htvBroken,
        Ichan_A, Ichan_B, Ichan_C, wp, buffer)
113  );

115 HTV_defect(htvID, htvC, htvBroken, Ichan_A, Ichan_B,
       Ichan_C) :=
116  htvBroken<htvID>.htvID().
117  ROBOT(htvID, htvC, htvBroken, Ichan_A, Ichan_B, Ichan_C);
```

```
119 MECHANIC(mtBroken, htvBroken) :=
120  ( mtBroken(id).tau.id<>.MECHANIC(mtBroken, htvBroken)
121  + htvBroken(id).tau.id<>.MECHANIC(mtBroken, htvBroken)
122  );
123 // Tool machines ////////////////////////////////

125 // One-place buffer
126 Buffer(channel, input, output) :=
127  channel(yes, no).yes<>.input<channel>.channel(wp).
128  BufferFull(channel, wp, input, output);

130 BufferFull(channel, wp, input, output) :=
131  channel(yes, no).no<>.output<channel>.channel<wp>.
132  Buffer(channel, input, output);

134 // Controller
135 Ctrl(channel, yes, no, input, output, empty, full) :=
136  channel<yes, no>.
137  ( yes().empty<input>.
138    Ctrl(channel, yes, no, input, output, empty, full)
139  + no().full<output>.
140    Ctrl(channel, yes, no, input, output, empty, full)
141  );

143 Handler(intern, extern) :=
144  intern(io).io(channel).extern<channel>.
145  Handler(intern, extern);

147 B(mtID, mtBroken,full, empty) :=
148  full(input).input(channel).
149  Work(mtID, mtBroken, full, empty, channel);

151 Work(mtID, mtBroken, full, empty, channel) :=
152  channel(wp).tau.
153  ( Put(mtID, mtBroken, full, empty, wp)
154  + mtBroken<mtID>.mtID().tau.
155    Put(mtID, mtBroken, full, empty, wp)
156  );

158 Put(mtID, mtBroken,full, empty, wp) :=
159  empty(output).output(channel).channel<wp>.
160  B(mtID, mtBroken, full, empty);

162 MT(mtID, address, htvC, mtBroken,
163    i, Iyes, Ino, Iempty, Ifull, Iinput, Ioutput,
164    o, Oyes, Ono, Oempty, Ofull, Oinput, Ooutput) :=
```

```
165 ( Buffer(i, Iinput, Ioutput)
166 | Buffer(o, Oinput, Ooutput)
167 | Ctrl(i, Iyes, Ino, Iinput, Ioutput, Iempty, Ifull)
168 | Ctrl(o, Oyes, Ono, Oinput, Ooutput, Oempty, Ofull)
169 | Handler(Iempty, address)
170 | Handler(Ofull, htvC)
171 | B(mtID, mtBroken, Ifull, Oempty)
172 );
```

B.2. PNCSA Protocol

Part of the PNCSA protocol (Standard Protocol for Connection to the Authorisation System)[1] was introduced as a Petri net case study in [Fin93] (see Fig. B.2) and has since been used in several publications, e.g. [BF99, Van04, GRV05, Gee07]. The textual representation of the Petri net in the format used by the MIST2 tool[2] is given in Listing B.2. The shortest transition sequence to reach the control-state CL2, AP, CR, FN, GTDT110 (marked grey in Fig. B.2) is t1, t19, t2, t20, t3, t21, t4, t5, t22, t26, t27, t30, t35, t6, t11, t17, t1, t19, t2, t20, t3, t21, t4, t9, t27, t10, t11, t25, t17, t1, t19, t2, t13, t17, t1 with a length of 35.

Listing B.2: Source of the PNCSA protocol

```
1  vars
2   LI1 CL1 ILC1 WCA WCC WAC
    WSTD1req WDT1 WDN1 WLI1
    WCL1 LI2 CL2 ILC2 WCR WCN
    WDT2 WSTD2req WDN2 WLI2
    WCL2 AP CA CR CC CN ON AC
    FN GTDT100 GTDT110
3  rules
4   # t1
5   ILC1>=1 ->
6    ILC1' = ILC1-1,
7    WCA' = WCA+1,
8    AP' = AP+1;

10  # t2
11  WCA>=1, CA>=1 ->
12   WCA' = WCA-1,
13   WCC' = WCC+1,
14   CA' = CA-1,
15   CR' = CR+1;

17  # t3
18  WCC>=1, CC>=1 ->
19   WCC' = WCC-1,
20   WAC' = WAC+1,
21   CC' = CC-1,
22   CN' = CN+1;

24  # t4
25  WAC>=1, AC>=1 ->
26   WAC' = WAC-1,
```

[1] Unfortunately, the publication from which it originated, [GIE88], wasn't available to the author.

[2] See `https://github.com/pierreganty/mist/wiki#input-format-of-mist`.

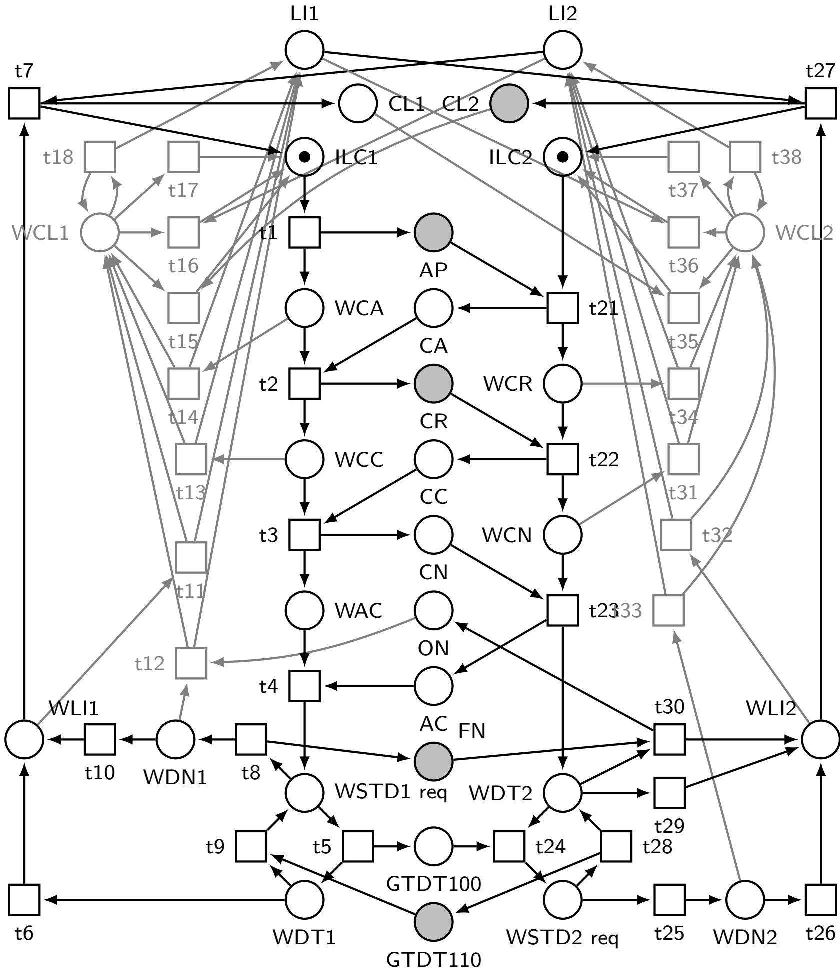


Figure B.2.: Considered part of the PNCSA protocol ([Fin93])

```
27      WSTD1req' = WSTD1req+1,
28      AC' = AC-1;

30    # t5
31    WSTD1req>=1 ->
32      WSTD1req' = WSTD1req-1,
33      WDT1' = WDT1+1,
34      GTDT100' = GTDT100+1;
```

```
36 # t6
37 WDT1>=1 ->
38  WDT1' = WDT1-1,
39  WLI1' = WLI1+1;

41 # t7
42 WLI1>=1, LI2>=1 ->
43  CL1' = CL1+1,
44  ILC1' = ILC1+1,
45  WLI1' = WLI1-1,
46  LI2' = LI2-1;

48 # t8
49 WSTD1req>=1 ->
50  WSTD1req' = WSTD1req-1,
51  WDN1' = WDN1+1,
52  FN' = FN+1;

54 # t9
55 WDT1>=1, GTDT110>=1 ->
56  WSTD1req' = WSTD1req+1,
57  WDT1' = WDT1-1,
58  GTDT110' = GTDT110-1;

60 # t10
61 WDN1>=1 ->
62  WDN1' = WDN1-1,
63  WLI1' = WLI1+1;

65 # t11
66 WLI1>=1 ->
67  LI1' = LI1+1,
68  WLI1' = WLI1-1,
69  WCL1' = WCL1+1;

71 # t12
72 WDN1>=1, ON>=1 ->
73  LI1' = LI1+1,
74  WDN1' = WDN1-1,
75  WCL1' = WCL1+1,
76  ON' = ON-1;

78 # t13
79 WCC>=1 ->
80  LI1' = LI1+1,
81  WCC' = WCC-1,
82  WCL1' = WCL1+1;

84 # t14
85 WCA>=1 ->
86  LI1' = LI1+1,
87  WCA' = WCA-1,
88  WCL1' = WCL1+1;

90 # t15
91 WCL1>=1, CL2>=1 ->
92  ILC1' = ILC1+1,
93  WCL1' = WCL1-1,
94  CL2' = CL2-1;

96 # t16
97 WCL1>=1, LI2>=1 ->
98  ILC1' = ILC1+1,
99  WCL1' = WCL1-1,
100 LI2' = LI2-1;

102 # t17
103 WCL1>=1 ->
104  ILC1' = ILC1+1,
105  WCL1' = WCL1-1;

107 # t18
108 WCL1>=1 ->
109  LI1' = LI1+1;

111 # t21
112 ILC2>=1, AP>=1 ->
113  ILC2' = ILC2-1,
114  WCR' = WCR+1,
115  AP' = AP-1,
116  CA' = CA+1;

118 # t22
119 WCR>=1, CR>=1 ->
120  WCR' = WCR-1,
121  WCN' = WCN+1,
122  CR' = CR-1,
123  CC' = CC+1;

125 # t23
126 WCN>=1, CN>=1 ->
127  WCN' = WCN-1,
128  WDT2' = WDT2+1,
```

```
129    CN' = CN-1,
130    AC' = AC+1;

132  # t24
133  WDT2>=1, GTDT100>=1 ->
134    WDT2' = WDT2-1,
135    WSTD2req' = WSTD2req+1,
136    GTDT100' = GTDT100-1;

138  # t25
139  WSTD2req>=1 ->
140    WSTD2req' = WSTD2req-1,
141    WDN2' = WDN2+1;

143  # t26
144  WDN2>=1 ->
145    WDN2' = WDN2-1,
146    WLI2' = WLI2+1;

148  # t27
149  LI1>=1, WLI2>=1 ->
150    LI1' = LI1-1,
151    CL2' = CL2+1,
152    ILC2' = ILC2+1,
153    WLI2' = WLI2-1;

155  # t28
156  WSTD2req>=1 ->
157    WDT2' = WDT2+1,
158    WSTD2req' = WSTD2req-1,
159    GTDT110' = GTDT110+1;

161  # t29
162  WDT2>=1 ->
163    WDT2' = WDT2-1,
164    WLI2' = WLI2+1;

166  # t30
167  WDT2>=1, FN>=1 ->
168    WDT2' = WDT2-1,
169    WLI2' = WLI2+1,
170    ON' = ON+1,
171    FN' = FN-1;

173  # t31
174  WCN>=1 ->
175    LI2' = LI2+1,
176    WCN' = WCN-1,
177    WCL2' = WCL2+1;

179  # t32
180  WLI2>=1 ->
181    LI2' = LI2+1,
182    WLI2' = WLI2-1,
183    WCL2' = WCL2+1;

185  # t33
186  WDN2>=1 ->
187    LI2' = LI2+1,
188    WDN2' = WDN2-1,
189    WCL2' = WCL2+1;

191  # t34
192  WCR>=1 ->
193    LI2' = LI2+1,
194    WCR' = WCR-1,
195    WCL2' = WCL2+1;

197  # t35
198  CL1>=1, WCL2>=1 ->
199    CL1' = CL1-1,
200    ILC2' = ILC2+1,
201    WCL2' = WCL2-1;

203  # t36
204  LI1>=1, WCL2>=1 ->
205    LI1' = LI1-1,
206    ILC2' = ILC2+1,
207    WCL2' = WCL2-1;

209  # t37
210  WCL2>=1 ->
211    ILC2' = ILC2+1,
212    WCL2' = WCL2-1;

214  # t38
215  WCL2>=1 ->
216    LI2' = LI2+1;

218  init
219   LI1=0, CL1=0, ILC1=1, WCA
      =0, WCC=0, WAC=0, WSTD1req
      =0, WDT1=0, WDN1=0, WLI1=0,
       WCL1=0, LI2=0, CL2=0, ILC2
```

```
    =1, WCR=0, WCN=0, WDT2=0,
    WSTD2req=0, WDN2=0, WLI2=0,
     WCL2=0, AP=0,CA=0, CR=0,
    CC=0, CN=0, ON=0, AC=0, FN
    =0, GTDT100=0, GTDT110=0

221 target
222  CL2>=1, AP>=1, CR>=1, FN
    >=1, GTDT110>=1
223 invariants
224  ILC1=1, WCA=1,WCC=1, WAC=1,
      WSTD1req=1, WDT1=1, WDN1
     =1, WLI1=1, WCL1=1
225  ILC2=1, WCR=1, WCN=1, WDT2
     =1, WSTD2req=1, WDN2=1,
     WLI2=1, WCL2=1
```

B.3. Kanban Production System

The Kanban production system from [MBC+95][3] consists of a fixed number of equally-structured cells depicted in Fig. B.3. A single cell models control flow of a single part of a production process. It contains a bulletin board containing cards which represent requests for assembly parts. Whenever a part is to enter the input storage of a Kanban cell, a card is taken from the bulletin board and attached to the part. Parts at the input storage are then processed sequentially and moved to the output storage. When exiting the output storage and being transported to another cell, the card is removed from the part and added to the bulletin board. Each cell owns a fixed number of cards. At any point in time, a cell contains at most parts as the number of cards it owns.

To model larger production systems, several Kanban cells can be interconnected. In our model, there is an infinite number of parts available to the first cell and we assume an infinite storage as an output of the last cell.

The cell design was modified for concurrent processing of parts in [CT96], where also the interconnection of four cells was introduced. Also, a reduction of certain transitions and places was proposed, resulting in cells as pictured in Fig. B.4. A modified cell checks if the production step for the currently processed part is finished or if the part is erroneous

[3] The specification from [MBC+95] was introduced in 1989 by G. Balbo and G. Fanceschinis: "Modelling flexible manufacturing systems with generalized stochastic Petri nets" as well as P. Legato, A. Bobbio, and L. Roberti: "The effect of failures and repairs on multiple cell production lines" which were not available to the author.

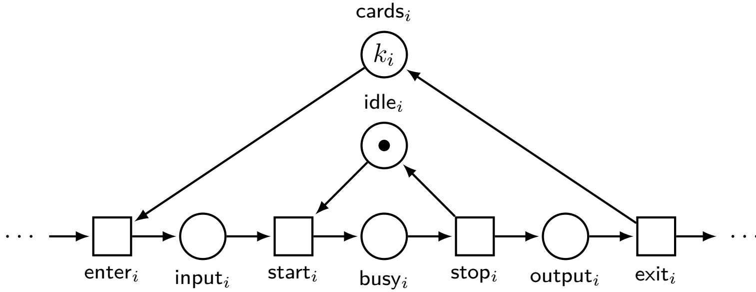


Figure B.3.: A Kanban production cell according to [MBC+95]

and it is decided whether the production step has to be repeated. Moreover, arbitrarily many parts can be processed concurrently. In [CM97] the model consisting of those reduced cells was analysed. It has parametrised markings for the bulletin boards and is shown in Fig. B.5.

The property to check asks, if a marking is reachable where at least four cards are located at bulletin boards of cells 2, 3, and 4 and two parts are at place check_2 of cell 2 and six parts are at place busy_4 of cell 4. A shortest trace of length 48 to such a marking starts with six cards in cells 1, 2, 3, and nine cards in cell 4. It is enter_1, ok_1, exit_1, ok_2, ok_3, enter_4, $(\mathsf{enter}_1, \mathsf{ok}_1, \mathsf{exit}_1)^3$, $(\mathsf{ok}_2, \mathsf{ok}_3, \mathsf{enter}_4, \mathsf{enter}_1, \mathsf{ok}_1, \mathsf{exit}_1)^4$, ok_2, ok_3, enter_4, (redo_4), putting six (additional) tokens at the bulletin board of cell 1 and two (additional) tokens on place check_3. The textual representation of the Petri net in the format used by the MIST2 tool[4] is given in Listing B.3.

[4] See `https://github.com/pierreganty/mist/wiki#input-format-of-mist`.

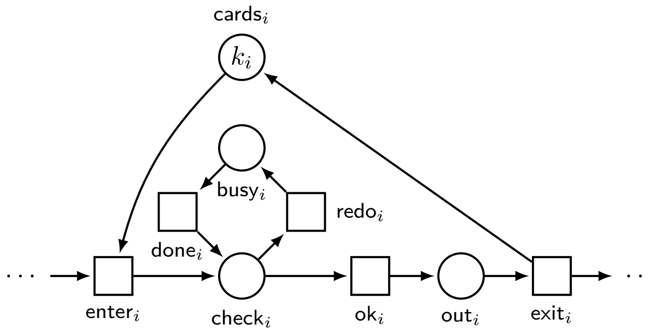


Figure B.4.: A Kanban production cell similar to [CM97]

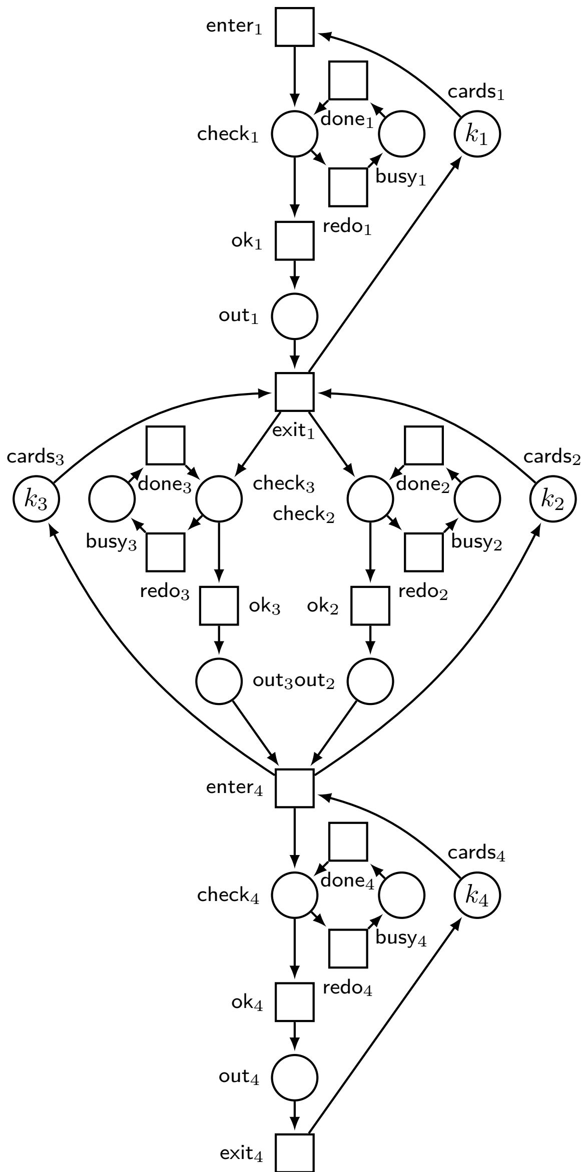


Figure B.5.: Considered Kanban production system ([CM97])

Listing B.3: Source of the Kanban production system

```
vars
 check1 busy1 cards1 out1
 check2 busy2 cards2 out2
 check3 busy3 cards3 out3
 check4 busy4 cards4 out4

rules
 # enter1
 cards1>=1 ->
  check1' = check1+1,
  cards1' = cards1-1;

 # redo1
 check1>=1 ->
  check1' = check1-1,
  busy1' = busy1+1;

 # done1
 busy1>=1 ->
  check1' = check1+1,
  busy1' = busy1-1;

 # ok1
 check1>=1 ->
  check1' = check1-1,
  out1' = out1+1;

 # exit1
 out1>=1, cards2>=1, cards3
 >=1 ->
  cards1' = cards1+1,
  out1' = out1-1,
  check2' = check2+1,
  cards2' = cards2-1,
  check3' = check3+1,
  cards3' = cards3-1;

 # redo2
 check2>=1 ->
  check2' = check2-1,
  busy2' = busy2+1;

 # done2
 busy2>=1 ->
  check2' = check2+1,
  busy2' = busy2-1;

 # ok2
 check2>=1 ->
  check2' = check2-1,
  out2' = out2+1;

 # enter4
 out2>=1, out3>=1, cards4>=1
  ->
  cards2' = cards2+1,
  out2' = out2-1,
  cards3' = cards3+1,
  out3' = out3-1,
  check4' = check4+1,
  cards4' = cards4-1;

 # redo3
 check3>=1 ->
  check3' = check3-1,
  busy3' = busy3+1;

 # done3
 busy3>=1 ->
  check3' = check3+1,
  busy3' = busy3-1;

 # ok3
 check3>=1 ->
  check3' = check3-1,
  out3' = out3+1;

 # redo4
 check4>=1 ->
  check4' = check4-1,
  busy4' = busy4+1;

 # done4
 busy4>=1 ->
  check4' = check4+1,
  busy4' = busy4-1;

 # ok4
 check4>=1 ->
  check4' = check4-1,
```

```
86      out4' = out4+1;

88    # exit4
89    out4>=1 ->
90      cards4' = cards4+1,
91      out4' = out4-1;

93  init
94    check1=0, busy1=0, cards1
      >=1, out1=0, check2=0,
      busy2=0, cards2>=1, out2=0,
        check3=0, busy3=0, cards3
      >=1, out3=0, check4=0,
      busy4=0, cards4>=1, out4=0

96  target
97    check2>=2, busy4>=6, cards2
      >=4, cards3>=4, cards4>=4

99  invariants
100   check3=1, busy3=1, cards2
      =1, out3=1
101   check3=1, busy3=1, cards3
      =1, out3=1
102   check2=1, busy2=1, cards2
      =1, out2=1
103   check2=1, busy2=1, cards3
      =1, out2=1
104   check4=1, busy4=1, cards4
      =1, out4=1
105   check1=1, busy1=1, cards1
      =1, out1=1
```

B.4. Delegate Buffer Program

In [Lea99, pp. 271–272], Lea gives a bounded queue using the *delegation pattern* (cf. [GHJ94]), i.e. calls to `put(Object)` and `take()` methods of the main class are delegated to helper classes which perform the actual operations.

The Java code of Lea's example is given in Listing B.4.

Here, the `put(Object)` method is meant to append the given object to the queue and block when the capacity is exceeded. Analogously, the `take()` method is intended to block until there are objects in the queue and then remove and return the oldest one. The queue is implemented as a circular buffer using the object array `array`. Lea observes that the helpers can be implemented as two instances of the same class `Exchanger`, offering method `exchange(Object)`. An `Exchanger` contains a number of available slots and a (circular) index to the next slot to process. Method `exchange(Object)` performs the actual exchange: blocking until at least one slot is available and then returning the contents of that slot while overwriting it with the given object. To implement `take()` method `exchange(null)` is called and to implement `put(o)` `exchange(o)` is called: Removing an object from the buffer inserts `null` while adding an object to the buffer returns `null`.

Synchronization is achieved by the `synchronization` keyword and by using `wait()` and `notify()` mechanics of monitor objects. To accomplish the blocking when the buffer is full and an object is to be added or the buffer is empty and `take()` is called, an `Exchanger` object calls `wait()` on itself when no slots are available. At the end of an `exchange(Object)` call by a helper object, the main class's method `removedSlotNotification(Exchanger)` is called which performs a case distinction to inform the other helper object by calling `addedSlotNotification()` that one more slot is available. In the `addedSlotNotification()` method, `notify()` is called on the object to awaken one `wait()`ing thread which then re-checks if slots are available.
Although it is commonly considered good practice to choose `notifyAll()` over `notify()`[5], Lea uses the `notify()` mechanism for performance reasons. Furthermore, Lea declares the monitor objects `putter` and `taker` (on which `wait()` is called) `private` and class `BoundedBufferWithDelegates` is declared `final`, prohibiting inheriting from this class.
L. Van Begin constructed a Petri net with transfer which represents the concurrent `BoundedBufferWithDelegates` program [Van04]. The Petri net is generated by Van Begin by utilizing predicate abstraction techniques in sense of [GS97, BMMR01, BPR02] to retrieve a concurrent boolean program which in turn is translated (with an intermediate step of a so-called *global machine*) into the given PNT . The textual form of L. Van Begin's Petri net with transfer in the format used by the MIST2 tool[6] is given in Listing B.5.
The unsafe control-state is both `putter` and `taker` assigning a value to the same `array` slot (both with `ptr == 0`) in line 60 of Listing B.4. It is not reachable.

Listing B.4: Java source of class `BoundedBufferWithDelegates` ([Lea99])

```
1 final class BoundedBufferWithDelegates {
2   private Object[] array;
3   private Exchanger putter;
4   private Exchanger taker;
```

[5]The `notify()` mechanism awakens *some* `wait()`ing thread for the respective monitor object `m`. Particularly in the case that object `m` is accessible (from outside the defining class) this poses a risk for adverse behaviour, e.g. deadlocks (cf. [Lea99, GBB$^+$06, Blo08]).

[6]See `https://github.com/pierreganty/mist/wiki#input-format-of-mist`.

```
6   public BoundedBufferWithDelegates(int capacity)
7    throws IllegalArgumentException {
8     if (capacity <= 0) throw new IllegalArgumentException();
9     array = new Object[capacity];
10    putter = new Exchanger(capacity);
11    taker = new Exchanger(0);
12  }

14  public void put(Object x) throws InterruptedException {
15    putter.exchange(x);
16  }

18  public Object take() throws InterruptedException {
19    return taker.exchange(null);
20  }

22  void removedSlotNotification(Exchanger h) { // relay
23    if (h == putter) taker.addedSlotNotification();
24    else             putter.addedSlotNotification();
25  }

27  protected class Exchanger { // inner class
28    protected int ptr = 0;      // circular index
29    protected int slots;        // number of usable slots
30    protected int waiting = 0; // number of waiting threads

32    Exchanger(int n) { slots = n; }

34    synchronized void addedSlotNotification() {
35      ++slots;
36      if (waiting > 0) // unblock a single waiting thread
37        notify();
38    }

40    Object exchange(Object x) throws InterruptedException {
41      Object old = null; // return value

43      synchronized(this) {
44        while (slots <= 0) { // wait for slots
45          ++waiting;
```

```
46            try {
47              wait();
48            }
49            catch(InterruptedException ie) {
50              notify();
51              throw ie;
52            }
53            finally {
54              --waiting;
55            }
56          }

58          --slots; // use slot
59          old = array[ptr];
60          array[ptr] = x;
61          ptr = (ptr + 1) % array.length; // advance position
62        }

64        removedSlotNotification(this); // notify of change
65        return old;
66      }
67    }
68  }
```

Listing B.5: Source of the delegate buffer program

```
1  vars
2  unlockT lockT unlockP lockP
   notslotTeq0 slotTeq0
   notslotTeq1 slotTeq1
   notslotTeq2  slotTeq2
   notslotPeq0 slotPeq0
   notslotPeq1 slotPeq1
   notslotPeq2  slotPeq2
   notptrTeq0 ptrTeq0
   notptrTeq1 ptrTeq1
   notptrPeq0 ptrPeq0
   notptrPeq1 ptrPeq1 put
   Pwhile P1 Pwait Pafterwait
   Pdecslot Pincptr Passign
   Pnotify Pslotinc
   Pbeforenotify Pafternotify
   Pend take Twhile T1 Twait
   Tafterwait Tdecslot Tincptr
   Tassign Tnotify Tslotinc
   Tbeforenotify Tafternotify
   Tend

4  rules
5    #putter
6    put>=1, unlockP>=1 ->
7      put'=put-1,
8      Pwhile'=Pwhile+1,
9      unlockP'=unlockP-1,
10     lockP'=lockP+1;

12   Pwhile>=1, slotPeq0>=1 ->
13     Pwhile'=Pwhile-1,
14     P1'=P1+1;
```

```
16  Pwhile>=1, notslotPeq1>=1,
     notslotPeq2>=1 ->
17    Pwhile'=Pwhile-1,
18    P1'=P1+1;

20  P1>=1, lockP>=1 ->
21    P1'=P1-1,
22    Pwait'=Pwait+1,
23    lockP'=lockP-1,
24    unlockP'=unlockP+1;

26  Pafterwait>=1, unlockP>=1
    ->
27    Pafterwait'=Pafterwait
      -1,
28    Pwhile'=Pwhile+1,
29    unlockP'=unlockP-1,
30    lockP'=lockP+1;

32  Pwhile>=1, notslotPeq0>=1
    ->
33    Pwhile'=Pwhile-1,
34    Pdecslot'=Pdecslot+1;

36  Pwhile>=1, notslotPeq1>=1,
     notslotPeq2>=1,
    notslotPeq0>=1 ->
37    Pwhile'=Pwhile-1,
38    Pdecslot'=Pdecslot+1;

40  Pdecslot>=1, slotPeq0>=1
    ->
41    Pdecslot'=Pdecslot-1,
42    Pincptr'=Pincptr+1,
43    slotPeq0'=slotPeq0-1,
44    notslotPeq0'=notslotPeq0
      +1;

46  Pdecslot>=1, slotPeq1>=1
    ->
47    Pdecslot'=Pdecslot-1,
48    Pincptr'=Pincptr+1,
49    slotPeq1'=slotPeq1-1,
50    notslotPeq1'=notslotPeq1
      +1,
51    notslotPeq0'=0,
52    slotPeq0'=slotPeq0+
      notslotPeq0+0;

54  Pdecslot>=1, slotPeq2>=1
    ->
55    Pdecslot'=Pdecslot-1,
56    Pincptr'=Pincptr+1,
57    slotPeq2'=slotPeq2-1,
58    notslotPeq2'=notslotPeq2
      +1,
59    notslotPeq1'=0,
60    slotPeq1'=slotPeq1+
      notslotPeq1+0;

62  Pdecslot>=1, notslotPeq0
    >=1, notslotPeq1>=1,
    notslotPeq2>=1 ->
63    Pdecslot'=Pdecslot-1,
64    Pincptr'=Pincptr+1;

66  Pdecslot>=1, notslotPeq0
    >=1, notslotPeq1>=1,
    notslotPeq2>=1 ->
67    Pdecslot'=Pdecslot-1,
68    Pincptr'=Pincptr+1,
69    notslotPeq2'=notslotPeq2
      -1,
70    slotPeq2'=slotPeq2+1;

72  Pincptr>=1, ptrPeq0>=1 ->
73    Pincptr'=Pincptr-1,
74    Passign'=Passign+1,
75    ptrPeq0'=ptrPeq0-1,
76    notptrPeq0'=notptrPeq0
      +1,
77    ptrPeq1'=ptrPeq1+
      notptrPeq1+0,
78    notptrPeq1'=0;

80  Pincptr>=1, ptrPeq1>=1 ->
81    Pincptr'=Pincptr-1,
82    Passign'=Passign+1,
83    ptrPeq1'=ptrPeq1-1,
84    notptrPeq1'=notptrPeq1
      +1,
85    ptrPeq0'=ptrPeq0+
      notptrPeq0+0,
86    notptrPeq0'=0;
```

```
88    Pincptr>=1, notptrPeq1>=1,
       notptrPeq0>=1 ->
89      Pincptr'=Pincptr-1,
90      Passign'=Passign+1,
91      ptrPeq0'=ptrPeq0+1,
92      notptrPeq0'=notptrPeq0
        -1;

94    Pincptr>=1, notptrPeq1>=1,
       notptrPeq0>=1 ->
95      Pincptr'=Pincptr-1,
96      Passign'=Passign+1,
97      ptrPeq1'=ptrPeq1+1,
98      notptrPeq1'=notptrPeq1
        -1;

100   Passign>=1, lockP>=1 ->
101     Passign'=Passign-1,
102     Pnotify'=Pnotify+1,
103     lockP'=lockP-1,
104     unlockP'=unlockP+1;

106   Pnotify>=1, unlockT>=1 ->
107     Pnotify'=Pnotify-1,
108     Pslotinc'=Pslotinc+1,
109     unlockT'=unlockT-1,
110     lockT'=lockT+1;

112   Pslotinc>=1, slotTeq0>=1
      ->
113     Pslotinc'=Pslotinc-1,
114     Pbeforenotify'=
        Pbeforenotify+1,
115     slotTeq0'=slotTeq0-1,
116     notslotTeq0'=notslotTeq0
        +1,
117     slotTeq1'=slotTeq1+
        notslotTeq1+0,
118     notslotTeq1'=0;

120   Pslotinc>=1, slotTeq1>=1
      ->
121     Pslotinc'=Pslotinc-1,
122     Pbeforenotify'=
        Pbeforenotify+1,
123     slotTeq1'=slotTeq1-1,
124     notslotTeq1'=notslotTeq1
        +1,
125     slotTeq2'=slotTeq2+
        notslotTeq2+0,
126     notslotTeq2'=0;

128   Pslotinc>=1, slotTeq2>=1
      ->
129     Pslotinc'=Pslotinc-1,
130     Pbeforenotify'=
        Pbeforenotify+1,
131     slotTeq2'=slotTeq2-1,
132     notslotTeq2'=notslotTeq2
        +1;

134   Pslotinc>=1, notslotTeq0
      >=1, notslotTeq1>=1,
      notslotTeq2>=1 ->
135     Pslotinc'=Pslotinc-1,
136     Pbeforenotify'=
        Pbeforenotify+1;

138   Pslotinc>=1, notslotTeq0
      >=1, notslotTeq1>=1,
      notslotTeq2>=1 ->
139     Pslotinc'=Pslotinc-1,
140     Pbeforenotify'=
        Pbeforenotify+1,
141     slotTeq0'=slotTeq0+1,
142     notslotTeq0'=notslotTeq0
        -1;

144   #attention, c'est un
      notify normalement!
145   Pbeforenotify>=1 ->
146     Pbeforenotify'=
        Pbeforenotify-1,
147     Pafternotify'=
        Pafternotify+1,
148     Pafterwait'=Pafterwait+
        Pwait+0,
149     Pwait'=0,
150     Tafterwait'=Tafterwait+
        Twait+0,
151     Twait'=0;

153   Pafternotify>=1, lockT>=1
```

```
    ->
154    Pafternotify'=
       Pafternotify-1,
155    Pend'=Pend+1,
156    lockT'=lockT-1,
157    unlockT'=unlockT+1;

159  Pend>=1 ->
160    Pend'=Pend-1,
161    put'=put+1;

163  #taker
164  take>=1, unlockT>=1 ->
165    take'=take-1,
166    Twhile'=Twhile+1,
167    unlockT'=unlockT-1,
168    lockT'=lockT+1;

170  Twhile>=1, slotTeq0>=1 ->
171    Twhile'=Twhile-1,
172    T1'=T1+1;

174  Twhile>=1, notslotTeq1>=1,
      notslotTeq2>=1 ->
175    Twhile'=Twhile-1,
176    T1'=T1+1;

178  T1>=1, lockT>=1 ->
179    T1'=T1-1,
180    Twait'=Twait+1,
181    lockT'=lockT-1,
182    unlockT'=unlockT+1;

184  Tafterwait>=1, unlockT>=1
     ->
185    Tafterwait'=Tafterwait
       -1,
186    Twhile'=Twhile+1,
187    unlockT'=unlockT-1,
188    lockT'=lockT+1;

190  Twhile>=1, notslotTeq0>=1
     ->
191    Twhile'=Twhile-1,
192    Tdecslot'=Tdecslot+1;

194  Twhile>=1, notslotTeq1>=1,
      notslotTeq2>=1,
     notslotTeq0>=1 ->
195    Twhile'=Twhile-1,
196    Tdecslot'=Tdecslot+1;


199  Tdecslot>=1, slotTeq0>=1
     ->
200    Tdecslot'=Tdecslot-1,
201    Tincptr'=Tincptr+1,
202    slotTeq0'=slotTeq0-1,
203    notslotTeq0'=notslotTeq0
       +1;

205  Tdecslot>=1, slotTeq1>=1
     ->
206    Tdecslot'=Tdecslot-1,
207    Tincptr'=Tincptr+1,
208    slotTeq1'=slotTeq1-1,
209    notslotTeq1'=notslotTeq1
       +1,
210    notslotTeq0'=0,
211    slotTeq0'=slotTeq0+
       notslotTeq0+0;

213  Tdecslot>=1, slotTeq2>=1
     ->
214    Tdecslot'=Tdecslot-1,
215    Tincptr'=Tincptr+1,
216    slotTeq2'=slotTeq2-1,
217    notslotTeq2'=notslotTeq2
       +1,
218    notslotTeq1'=0,
219    slotTeq1'=slotTeq1+
       notslotTeq1+0;

221  Tdecslot>=1, notslotTeq0
     >=1, notslotTeq1>=1,
     notslotTeq2>=1 ->
222    Tdecslot'=Tdecslot-1,
223    Tincptr'=Tincptr+1;

225  Tdecslot>=1, notslotTeq0
     >=1, notslotTeq1>=1,
     notslotTeq2>=1 ->
226    Tdecslot'=Tdecslot-1,
227    Tincptr'=Tincptr+1,
```

```
228       notslotTeq2'=notslotTeq2
          -1,
229       slotTeq2'=slotTeq2+1;


232     Tincptr>=1, ptrTeq0>=1 ->
233       Tincptr'=Tincptr-1,
234       Tassign'=Tassign+1,
235       ptrTeq0'=ptrTeq0-1,
236       notptrTeq0'=notptrTeq0
          +1,
237       ptrTeq1'=ptrTeq1+
          notptrTeq1+0,
238       notptrTeq1'=0;

240     Tincptr>=1, ptrTeq1>=1 ->
241       Tincptr'=Tincptr-1,
242       Tassign'=Tassign+1,
243       ptrTeq1'=ptrTeq1-1,
244       notptrTeq1'=notptrTeq1
          +1,
245       ptrTeq0'=ptrTeq0+
          notptrTeq0+0,
246       notptrTeq0'=0;

248     Tincptr>=1, notptrTeq1>=1,
         notptrTeq0>=1 ->
249       Tincptr'=Tincptr-1,
250       Tassign'=Tassign+1,
251       ptrTeq0'=ptrTeq0+1,
252       notptrTeq0'=notptrTeq0
          -1;

254     Tincptr>=1, notptrTeq1>=1,
         notptrTeq0>=1 ->
255       Tincptr'=Tincptr-1,
256       Tassign'=Tassign+1,
257       ptrTeq1'=ptrTeq1+1,
258       notptrTeq1'=notptrTeq1
          -1;


261     Tassign>=1, lockT>=1 ->
262       Tassign'=Tassign-1,
263       Tnotify'=Tnotify+1,
264       lockT'=lockT-1,
265       unlockT'=unlockT+1;

267     Tnotify>=1, unlockP>=1 ->
268       Tnotify'=Tnotify-1,
269       Tslotinc'=Tslotinc+1,
270       unlockP'=unlockP-1,
271       lockP'=lockP+1;

273     Tslotinc>=1, slotPeq0>=1
        ->
274       Tslotinc'=Tslotinc-1,
275       Tbeforenotify'=
          Tbeforenotify+1,
276       slotPeq0'=slotPeq0-1,
277       notslotPeq0'=notslotPeq0
          +1,
278       slotPeq1'=slotPeq1+
          notslotPeq1+0,
279       notslotPeq1'=0;


282     Tslotinc>=1, slotPeq1>=1
        ->
283       Tslotinc'=Tslotinc-1,
284       Tbeforenotify'=
          Tbeforenotify+1,
285       slotPeq1'=slotPeq1-1,
286       notslotPeq1'=notslotPeq1
          +1,
287       slotPeq2'=slotPeq2+
          notslotPeq2+0,
288       notslotPeq2'=0;

290     Tslotinc>=1, slotPeq2>=1
        ->
291       Tslotinc'=Tslotinc-1,
292       Tbeforenotify'=
          Tbeforenotify+1,
293       slotPeq2'=slotPeq2-1,
294       notslotPeq2'=notslotPeq2
          +1;

296     Tslotinc>=1, notslotPeq0
        >=1, notslotPeq1>=1,
        notslotPeq2>=1 ->
297       Tslotinc'=Tslotinc-1,
298       Tbeforenotify'=
          Tbeforenotify+1;
```

```
300     Tslotinc>=1, notslotPeq0
        >=1, notslotPeq1>=1,
        notslotPeq2>=1 ->
301         Tslotinc'=Tslotinc-1,
302         Tbeforenotify'=
            Tbeforenotify+1,
303         slotPeq0'=slotPeq0+1,
304         notslotPeq0'=notslotPeq0
            -1;

306     #attention, c'est un
        notify normalement!
307     Tbeforenotify>=1 ->
308         Tbeforenotify'=
            Tbeforenotify-1,
309         Tafternotify'=
            Tafternotify+1,
310         Pafterwait'=Pafterwait+
            Pwait+0,
311         Pwait'=0,
312         Tafterwait'=Tafterwait+
            Twait+0,
313         Twait'=0;

315     Tafternotify>=1, lockP>=1
        ->
316         Tafternotify'=
            Tafternotify-1,
317         Tend'=Tend+1,
318         lockP'=lockP-1,
319         unlockP'=unlockP+1;

321     Tend>=1 ->
322         Tend'=Tend-1,
323         take'=take+1;

325 init
326 unlockT=1, lockT=0, unlockP
    =1, lockP=0, notslotTeq0=0,
    slotTeq0=1, notslotTeq1=1,
    slotTeq1=0, notslotTeq2=1,
    slotTeq2=0, notslotPeq0=1,
    slotPeq0=0, notslotPeq1=1,
    slotPeq1=0, notslotPeq2=0,
    slotPeq2=1, notptrTeq0=0,
    ptrTeq0=1, notptrTeq1=1,
    ptrTeq1=0, notptrPeq0=0,
    ptrPeq0=1, notptrPeq1=1,
    ptrPeq1=0, put>=1, Pwhile=0,
    P1=0, Pwait=0, Pafterwait
    =0, Pdecslot=0, Pincptr=0,
    Passign=0, Pnotify=0,
    Pslotinc=0, Pbeforenotify=0,
    Pend=0, take>=1, Twhile=0,
    T1=0, Twait=0, Tafterwait=0,
    Tdecslot=0, Tincptr=0,
    Tassign=0, Tnotify=0,
    Tslotinc=0, Tbeforenotify=0,
    Tend=0, Pafternotify=0,
    Tafternotify=0

328 target
329 Passign>=1, ptrPeq0>=1,
    Tassign>=1, ptrTeq0>=1
330 #Passign>=1, ptrPeq1>=1,
    Tassign>=1, ptrTeq1>=1
331 #notslotTeq0>=1,
    notslotTeq1>=1, notslotTeq2
    >=1
332 #notslotPeq0>=1,
    notslotPeq1>=1, notslotPeq2
    >=1


335 invariants
336 unlockT=1, lockT=1
337 unlockP=1, lockP=1
338 notslotTeq0=1, slotTeq0=1
339 notslotTeq1=1, slotTeq1=1
340 notslotTeq2=1, slotTeq2=1
341 notslotPeq0=1, slotPeq0=1
342 notslotPeq1=1, slotPeq1=1
343 notslotPeq2=1, slotPeq2=1
344 notptrTeq0=1, ptrTeq0=1
345 notptrTeq1=1, ptrTeq1=1
346 notptrPeq0=1, ptrPeq0=1
347 notptrPeq1=1, ptrPeq1=1
348 unlockT=1, Pslotinc=1,
    Pbeforenotify=1,
    Pafternotify=1, Twhile=1, T1
    =1, Tdecslot=1, Tincptr=1,
    Tassign=1
349 unlockP=1, Pwhile=1, P1=1,
```

```
Pdecslot=1, Pincptr=1,          Tbeforenotify=1,
Passign=1, Tslotinc=1,          Tafternotify=1
```

APPENDIX C

Benchmark Data

Contents

C.1 Data Structures 266
C.2 Search Space Constructions and Search Guidance . . 269
C.3 Tool Comparison 286

We showed visual representations of the different effects on the runtime of our tools and compared several tools that solve coverability problems in the previous part of the thesis. In this appendix, we give most of the hard numbers that were the foundation of the visualizations.

C.1. Data Structures

In this section we present most of the benchmark results that went into the preliminary experiments in Ch. 6 on p. 160 (see Sect. 6.6.3 on p. 184 and Sect. 6.7 on p. 185 for the experimental set-up).
The tables contain the following columns (time measured in milliseconds).

H The height of the powerset search tree (PST) used to partition the set of states.

Model Name of the PN or PNT model, i.e. a coverability problem.

Iter. Number of successful benchmark iterations.

Max. ms Maximal time the tool needed to solve the coverability problem.

Mean ms Mean time the tool needed to solve the coverability problem.

σ **ms** Standard deviation of the time the tool needed to solve the coverability problem.

Max. M Maximal number of markings the tool created while solving the coverability problem.

Mean M Mean number of markings the tool created while solving the coverability problem.

σ **M** Standard deviation of the number of markings the tool created while solving the coverability problem.

For the first table, the number of iterations was fixed to 300.

H	Max. ms	Mean ms	σ ms	Max. M	Mean M	σ M
1	$3.43 \cdot 10^5$	$2.17 \cdot 10^5$	23,894.87	$5.38 \cdot 10^5$	$4.56 \cdot 10^5$	16,240.78
2	$3 \cdot 10^5$	$1.9 \cdot 10^5$	24,044.85	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	18,604.33
3	$2.45 \cdot 10^5$	$1.65 \cdot 10^5$	19,404.91	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	17,471.15
4	$2.06 \cdot 10^5$	$1.34 \cdot 10^5$	15,129.94	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,267.07
5	$1.91 \cdot 10^5$	$1.15 \cdot 10^5$	15,952.62	$5.38 \cdot 10^5$	$4.6 \cdot 10^5$	19,094.31
6	$1.36 \cdot 10^5$	96,895.87	11,920.55	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	17,802.1
7	$1.22 \cdot 10^5$	80,262.15	10,361.3	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	16,610.07
8	92,298.02	65,299.04	9,731.34	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	16,694.63
9	78,112.16	55,419.82	9,022.25	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	16,616.51
10	76,180.49	46,505.28	8,218.19	$5.39 \cdot 10^5$	$4.59 \cdot 10^5$	18,563.67
11	63,104.64	38,979.32	7,084.86	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	16,737.31
12	51,223.81	31,976.84	5,948.33	$4.98 \cdot 10^5$	$4.56 \cdot 10^5$	15,343.26
13	42,237.65	27,396.5	4,632.5	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	17,097.5
14	37,084.89	24,152.01	4,359.24	$5.39 \cdot 10^5$	$4.59 \cdot 10^5$	18,846.45
15	38,144.6	21,485.11	3,670.95	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	18,593.41
16	32,755.77	19,605.69	3,222.49	$5.38 \cdot 10^5$	$4.57 \cdot 10^5$	18,272.59
17	27,177.92	18,490.23	2,797.27	$5.38 \cdot 10^5$	$4.6 \cdot 10^5$	18,622.07
18	29,047.25	17,564.43	2,563.93	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	16,038.01
19	24,119.68	17,156.03	2,215.33	$5.38 \cdot 10^5$	$4.57 \cdot 10^5$	15,924.13
20	24,430.71	16,506.66	2,149.13	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	18,067.07
21	22,423	16,503.31	2,123.74	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,276.64
22	21,290.02	16,169.51	1,868.53	$5.38 \cdot 10^5$	$4.59 \cdot 10^5$	18,051.33
23	24,315.9	16,043.69	2,002.39	$5.38 \cdot 10^5$	$4.57 \cdot 10^5$	17,120.95
24	22,417.36	16,199.76	2,060.51	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,958.66
25	22,099.42	15,474.58	1,739.2	$5.38 \cdot 10^5$	$4.56 \cdot 10^5$	14,915.89
26	22,680.02	16,324.54	1,980.95	$4.99 \cdot 10^5$	$4.59 \cdot 10^5$	17,573.03
27	22,566.83	16,097.56	1,957.2	$5.39 \cdot 10^5$	$4.59 \cdot 10^5$	19,880.58
28	23,001.33	15,898.82	1,973.4	$5.38 \cdot 10^5$	$4.57 \cdot 10^5$	17,053.11
29	21,282.33	16,018.94	1,862.77	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,837.36
30	24,723.81	16,071.9	2,078.95	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,734.37
31	21,625.47	16,278.1	1,957.39	$5.39 \cdot 10^5$	$4.59 \cdot 10^5$	18,244.58
32	22,632.61	15,903.08	2,006.55	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	16,707.19
33	27,101.73	16,139	2,047.25	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	17,991.03
34	24,011.62	16,073.84	2,057.46	$5.38 \cdot 10^5$	$4.6 \cdot 10^5$	18,981.53
35	24,138.18	16,334.95	1,917.85	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	19,336.96
36	22,595.08	15,846.97	1,894.37	$4.98 \cdot 10^5$	$4.55 \cdot 10^5$	13,974
37	28,047.21	16,048.97	2,059.14	$5.39 \cdot 10^5$	$4.57 \cdot 10^5$	17,943.67
38	22,725.02	15,920.65	1,943.33	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,955.94
39	23,047.49	15,828.31	1,972.08	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	18,690.29
40	25,454.77	16,198.34	2,265.97	$5.38 \cdot 10^5$	$4.6 \cdot 10^5$	18,956.85
41	22,034.04	15,893.65	1,896.83	$4.99 \cdot 10^5$	$4.57 \cdot 10^5$	15,679.64
42	25,266.77	15,708.97	1,851.09	$4.99 \cdot 10^5$	$4.58 \cdot 10^5$	16,784.64
43	23,156.55	15,888.59	2,064.3	$5.38 \cdot 10^5$	$4.59 \cdot 10^5$	17,826.37
44	22,170.93	15,470.33	1,862.09	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,921.17
45	23,531.76	15,678.14	1,956.07	$5.39 \cdot 10^5$	$4.59 \cdot 10^5$	19,558.81
46	22,332	16,017.05	1,929.93	$5.38 \cdot 10^5$	$4.59 \cdot 10^5$	18,350.94
47	23,501.83	15,938.26	2,004.24	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	17,468.82
48	24,108.64	16,396.15	1,890.35	$5.38 \cdot 10^5$	$4.58 \cdot 10^5$	18,068.4
49	24,425.97	15,945.96	1,973.65	$5.39 \cdot 10^5$	$4.58 \cdot 10^5$	17,911.98
50	30,488.89	16,546.44	2,131.68	$5.38 \cdot 10^5$	$4.57 \cdot 10^5$	17,726.62
60	24,017.2	15,888.26	1,931.95	$4.99 \cdot 10^5$	$4.57 \cdot 10^5$	16,223.33

Table C.1.: Benchmark data for Tree Height in Sect. 6.6.3

Model	Iter.	Mean ms	σ ms	Mean M	σ M
basic-me	$7.47 \cdot 10^7$	$5.99 \cdot 10^{-2}$	9.53	3	0
basicextransfer	$5.91 \cdot 10^7$	$9.58 \cdot 10^{-2}$	3.39	11	0
lifo	$4.29 \cdot 10^7$	0.1	5.48	6	0
kanban	$1.02 \cdot 10^7$	0.58	4.12	85	0.82
java2	$6.39 \cdot 10^6$	0.81	11.11	8	0
moesi5	$6.11 \cdot 10^6$	0.83	13.77	15	0
pncsacover	$8.2 \cdot 10^5$	8.57	1.52	807.85	7.36
leaconflictset	$3.16 \cdot 10^5$	22.45	1.97	3,913.03	113.1
glotter	74,454	96.03	13.13	10,186.25	1,044.36
examplelea	69,699	102.98	21.58	12,314.12	2,090.46
queuedbusyflag	7,124	1,009.95	109.64	$1.33 \cdot 10^5$	139.35
java2.11.2	6,763	1,062.54	125.09	82,729.04	474.31
delegatebuffer.16.1	551	13,028.11	1,703.4	$4.59 \cdot 10^5$	19,346.13
hts	59	$1.69 \cdot 10^5$	$6.29 \cdot 10^5$	$3.8 \cdot 10^5$	$3.35 \cdot 10^5$

Table C.2.: Benchmark data for f_Σ before f_{supp} in Sect. 6.7.1

Model	Iter.	Mean ms	σ ms	Mean M	σ M
basic-me	$7.47 \cdot 10^7$	$9.89 \cdot 10^{-2}$	2.88	3	0
basicextransfer	$5.91 \cdot 10^7$	0.18	2.74	11	0
lifo	$4.29 \cdot 10^7$	0.16	5.09	6	0
kanban	$1.02 \cdot 10^7$	0.74	4.36	85	0.82
java2	$6.39 \cdot 10^6$	1.15	9.19	8	0
moesi5	$6.11 \cdot 10^6$	1.42	26.03	15	0
pncsacover	$8.2 \cdot 10^5$	14.64	2.16	807.84	7.37
leaconflictset	$3.16 \cdot 10^5$	38.54	4.31	3,888.62	92.66
glotter	74,454	131.78	25.18	9,661.51	1,505.86
examplelea	69,699	205.28	54.1	12,678.54	2,499.49
queuedbusyflag	7,124	1,851.29	280.69	$1.33 \cdot 10^5$	115.44
java2.11.2	6,763	2,181.47	214.95	$1.14 \cdot 10^5$	1,931.87
delegatebuffer.16.1	551	21,307.79	3,216.28	$5.38 \cdot 10^5$	49,218.13
hts	59	$4.25 \cdot 10^5$	68,494.69	$3.3 \cdot 10^5$	43,039.18

Table C.3.: Benchmark data for f_{supp} before f_Σ in Sect. 6.7.1

C.2. Search Space Constructions and Search Guidance

In this section we present most of the benchmark results that went into Sect. 7.2.3 in which we compare the different combinations of search space constructions (SSCs) and optimizations we discussed in Ch. 5 on p. 130. See Sect. 7.2 on p. 207 for the experimental set-up.
The tables contain the following columns (time measured in milliseconds).

Model Name of the PN or PNT model, i.e. a coverability problem.

Iter. Number of successful benchmark iterations.

Max. ms Maximal time the tool needed to solve the coverability problem.

Med. ms Median time the tool needed to solve the coverability problem.

σ **ms** Standard deviation of the time the tool needed to solve the coverability problem.

Med. T Median length of the found trace if a target state was coverable (else the table cell reads NaN). As the lengths of the found traces depend on the model's representation for each specific tool, they are not directly comparable but give a rough estimate.

If no data was collected due to a coverability problem not being solved within the time constraint of 5 minutes, each cell in the row for that model reads NaN.
The different SSCs and optimizations are

A acceleration,

G search guidance via syntactic distance and weight,

I pruning via inequality invariants, and

P partial-order reduction.

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	60,818	3.1	$6.03 \cdot 10^{-2}$	$1.27 \cdot 10^{-2}$	NaN
basicextransfer	$2.27 \cdot 10^5$	587.76	0.12	1.35	NaN
bingham-h250	135	7,423.04	2,577.01	2,299.28	NaN
consprod	$2.1 \cdot 10^5$	231.79	2.1	1.63	NaN
consprod2	59,633	236.16	$6.06 \cdot 10^{-2}$	0.97	NaN
csm	60,481	13.32	$2.62 \cdot 10^{-2}$	$6.42 \cdot 10^{-2}$	NaN
csm-broad	$2.75 \cdot 10^5$	271.05	1.22	1.08	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	238	5,001.24	3,758.76	307.05	17
delegatebuffer.16.1	79	14,978.31	11,304.91	1,130.58	18
efm	$4.05 \cdot 10^5$	204.53	0.25	0.64	NaN
examplelea	85	28,028.86	10,091.62	9,088.48	NaN
ext-rw	59,411	11.36	$6.99 \cdot 10^{-2}$	$7.32 \cdot 10^{-2}$	NaN
ext-rw-smallconsts	59,669	812.06	$6.92 \cdot 10^{-2}$	3.32	NaN
fms	$1.17 \cdot 10^5$	10.92	$3.04 \cdot 10^{-2}$	$6.18 \cdot 10^{-2}$	NaN
fms2	$1.16 \cdot 10^5$	11.61	$7.44 \cdot 10^{-2}$	$6.92 \cdot 10^{-2}$	NaN
german	$2.03 \cdot 10^5$	721.04	0.11	1.84	NaN
glotter	21,852	146.02	30.42	10.28	7
hts	12	72,474.41	56,490.6	5,932.95	29
java	1,989	706.3	448.63	44.88	15
java.10.0	1,509	899.42	587.91	58.65	14
java.11.0	3,980	384.34	220.27	18.77	15
java2	189	6,903.47	4,716.23	582.2	NaN
java2.10.2	1,375	914.04	648.83	52.4	14
java2.11.2	1,249	1,203.36	710.53	66.53	15
kanban	$1.22 \cdot 10^6$	1,253.8	0.44	3.63	109
km-nonterm.4.3	$4.11 \cdot 10^5$	226.8	0.21	0.55	18
km-nonterm.5.4	$3.93 \cdot 10^5$	268.94	0.3	0.66	28
km-nonterm.6.5	$3.75 \cdot 10^5$	241.52	0.41	0.6	40
lamport	60,507	5.66	$5.41 \cdot 10^{-2}$	$2.87 \cdot 10^{-2}$	NaN
leabasicapproach	59,748	239.36	0.14	0.98	5
leaconflictset	33,197	54.15	25.61	1.8	20
lifo	$4.33 \cdot 10^5$	277.4	$4.22 \cdot 10^{-2}$	0.79	NaN
manufacturing	60,630	1,070.22	$4.83 \cdot 10^{-2}$	4.35	NaN
mesh2x2	$1.12 \cdot 10^5$	12.78	$4.21 \cdot 10^{-2}$	$9.46 \cdot 10^{-2}$	NaN
mesh3x2	$1.6 \cdot 10^5$	470.92	$5.95 \cdot 10^{-2}$	1.19	NaN
moesi	$1.18 \cdot 10^5$	225.37	$4.46 \cdot 10^{-2}$	0.66	NaN
moesi5	$1.36 \cdot 10^5$	258.66	0.12	1.19	NaN
multi-me	60,463	883.59	$6.15 \cdot 10^{-2}$	4.44	NaN
multipoll	59,941	9.66	$3.72 \cdot 10^{-2}$	$5.77 \cdot 10^{-2}$	NaN
newdekker	$1.7 \cdot 10^5$	103.35	2.77	2.74	NaN
newrtp	60,610	6.82	$2.85 \cdot 10^{-2}$	$2.77 \cdot 10^{-2}$	NaN
peterson	60,030	979.29	$9.12 \cdot 10^{-2}$	4.71	NaN
pncsacover	$1.32 \cdot 10^5$	204.68	4.16	1.84	35
pncsasemiliv	$1.1 \cdot 10^5$	436.09	0.19	1.97	10
queuedbusyflag	110	10,463.62	8,068.63	684.9	NaN
read-write	$1.11 \cdot 10^5$	570.01	0.84	1.91	NaN
simplejavaexample	$1.26 \cdot 10^5$	99.23	4.41	1.12	13
transthesis	14,062	71.69	46.15	2.84	NaN

Table C.4.: Benchmark data for AGIP in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	60,757	0.35	$6.08 \cdot 10^{-2}$	$3.34 \cdot 10^{-3}$	NaN
basicextransfer	$2.26 \cdot 10^{5}$	623.05	0.12	1.46	NaN
bingham-h250	134	7,720.72	2,476.63	2,383	NaN
consprod	$2.11 \cdot 10^{5}$	246.77	2.08	1.66	NaN
consprod2	$1.16 \cdot 10^{5}$	244.68	$6.25 \cdot 10^{-2}$	0.72	NaN
csm	60,338	9.64	$2.59 \cdot 10^{-2}$	$3.92 \cdot 10^{-2}$	NaN
csm-broad	$2.75 \cdot 10^{5}$	301.07	1.21	1.15	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	239	4,854.8	3,745.03	320.17	17
delegatebuffer.16.1	72	16,595.37	12,345.44	1,374.45	18
efm	$4.07 \cdot 10^{5}$	455.35	0.25	0.94	NaN
examplelea	82	26,121.89	14,397.02	8,832.12	NaN
ext-rw	59,163	8.02	$7.03 \cdot 10^{-2}$	$4.65 \cdot 10^{-2}$	NaN
ext-rw-smallconsts	59,793	12.63	$6.64 \cdot 10^{-2}$	$8.06 \cdot 10^{-2}$	NaN
fms	$1.17 \cdot 10^{5}$	10.55	$2.95 \cdot 10^{-2}$	$6.09 \cdot 10^{-2}$	NaN
fms2	$1.16 \cdot 10^{5}$	482.53	$7.4 \cdot 10^{-2}$	2.12	NaN
german	$2.03 \cdot 10^{5}$	372.2	0.11	1.26	NaN
glotter	21,742	108.22	30.36	10.15	7
hts	11	80,187.69	63,681.68	7,497.49	29
java	1,917	782.02	464.26	46.8	15
java.10.0	1,514	816.03	588.7	46.19	14
java.11.0	3,950	428.61	221.76	19.71	15
java2	192	7,520.69	4,670.56	612.45	NaN
java2.10.2	1,348	904.52	661.35	55.93	14
java2.11.2	1,211	1,060.76	735.93	64.06	15
kanban	$3.73 \cdot 10^{5}$	26.67	2.13	0.46	81
km-nonterm.4.3	$4.09 \cdot 10^{5}$	178.62	0.22	0.58	18
km-nonterm.5.4	$3.91 \cdot 10^{5}$	1,017.63	0.3	1.75	28
km-nonterm.6.5	$3.81 \cdot 10^{5}$	226.78	0.36	0.56	40
lamport	60,437	8.07	$5.3 \cdot 10^{-2}$	$3.71 \cdot 10^{-2}$	NaN
leabasicapproach	59,843	237.68	0.14	0.97	5
leaconflictset	33,551	46.7	25.32	1.87	20
lifo	$2.25 \cdot 10^{5}$	703.1	$6.22 \cdot 10^{-2}$	1.56	NaN
manufacturing	60,688	5.4	$4.9 \cdot 10^{-2}$	$2.25 \cdot 10^{-2}$	NaN
mesh2x2	$1.1 \cdot 10^{5}$	1,887.74	$4.53 \cdot 10^{-2}$	5.68	NaN
mesh3x2	$1.6 \cdot 10^{5}$	472.35	$6.03 \cdot 10^{-2}$	1.19	NaN
moesi	$1.17 \cdot 10^{5}$	225.18	$4.46 \cdot 10^{-2}$	0.66	NaN
moesi5	$1.37 \cdot 10^{5}$	238.21	0.13	0.66	NaN
multi-me	60,494	7.66	$6.32 \cdot 10^{-2}$	$3.58 \cdot 10^{-2}$	NaN
multipoll	$1.17 \cdot 10^{5}$	880.49	$3.62 \cdot 10^{-2}$	2.94	NaN
newdekker	$1.68 \cdot 10^{5}$	70.79	2.79	2.67	NaN
newrtp	60,825	5.17	$2.76 \cdot 10^{-2}$	$2.11 \cdot 10^{-2}$	NaN
peterson	60,205	694.1	$8.85 \cdot 10^{-2}$	2.83	NaN
pncsacover	$1.41 \cdot 10^{5}$	267.27	3.75	2.23	35
pncsasemiliv	$1.1 \cdot 10^{5}$	620.7	0.19	3	10
queuedbusyflag	112	9,554.87	8,112.61	683	NaN
read-write	$1.11 \cdot 10^{5}$	937.6	0.82	3.07	NaN
simplejavaexample	$1.26 \cdot 10^{5}$	189.29	4.38	1.69	13
transthesis	11,970	86.29	56.36	4.38	NaN

Table C.5.: Benchmark data for GIP in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	60,754	3.71	$5.9 \cdot 10^{-2}$	$1.51 \cdot 10^{-2}$	NaN
basicextransfer	$2.24 \cdot 10^{5}$	247.68	0.15	0.69	NaN
bingham-h250	128	7,717.41	6,619.38	2,352.25	NaN
consprod	$2.12 \cdot 10^{5}$	177.54	2.06	1.29	NaN
consprod2	60,016	230.8	$5.92 \cdot 10^{-2}$	0.94	NaN
csm	60,327	11.43	$2.73 \cdot 10^{-2}$	$6.29 \cdot 10^{-2}$	NaN
csm-broad	$2.69 \cdot 10^{5}$	378.59	1.27	1.45	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	241	5,668.96	3,701.45	372.72	17
delegatebuffer.16.1	81	13,517.55	11,164.16	1,010.94	18
efm	$4.07 \cdot 10^{5}$	343.5	0.24	0.77	NaN
examplelea	44	39,168.67	26,513.23	12,881.34	NaN
ext-rw	59,417	9.74	$7 \cdot 10^{-2}$	$6.37 \cdot 10^{-2}$	NaN
ext-rw-smallconsts	59,042	11.2	$7.16 \cdot 10^{-2}$	$6.84 \cdot 10^{-2}$	NaN
fms	$1.17 \cdot 10^{5}$	15.65	$2.99 \cdot 10^{-2}$	$8.95 \cdot 10^{-2}$	NaN
fms2	$1.16 \cdot 10^{5}$	554.55	$7.6 \cdot 10^{-2}$	1.63	NaN
german	$2.01 \cdot 10^{5}$	735.28	0.11	1.89	NaN
glotter	20,231	263.58	33.47	11.34	7
hts	7	$1.42 \cdot 10^{5}$	$1.28 \cdot 10^{5}$	8,160.96	29
java	1,999	692.47	445.77	45.83	15
java.10.0	1,609	802.97	554.72	46.81	14
java.11.0	4,057	386.09	215.97	18.97	15
java2	196	6,456.47	4,538.38	548.55	NaN
java2.10.2	1,320	1,028.27	673.03	69.43	14
java2.11.2	1,217	993.57	732.99	60.72	15
kanban	$3.67 \cdot 10^{5}$	35.68	2.1	0.52	110
km-nonterm.4.3	$2.17 \cdot 10^{5}$	481.21	0.26	1.27	18
km-nonterm.5.4	$3.92 \cdot 10^{5}$	194.5	0.31	0.67	28
km-nonterm.6.5	$3.75 \cdot 10^{5}$	132.81	0.4	0.49	40
lamport	60,519	5.57	$5.33 \cdot 10^{-2}$	$2.88 \cdot 10^{-2}$	NaN
leabasicapproach	59,641	6.33	0.15	$3.58 \cdot 10^{-2}$	5
leaconflictset	33,910	51.03	25.05	1.73	20
lifo	$2.24 \cdot 10^{5}$	247.99	$6.02 \cdot 10^{-2}$	0.53	NaN
manufacturing	60,487	1,670.19	$5.46 \cdot 10^{-2}$	6.79	NaN
mesh2x2	$1.12 \cdot 10^{5}$	13.89	$4.41 \cdot 10^{-2}$	0.11	NaN
mesh3x2	$1.61 \cdot 10^{5}$	26.68	$6.13 \cdot 10^{-2}$	0.24	NaN
moesi	$1.16 \cdot 10^{5}$	233.62	$4.66 \cdot 10^{-2}$	0.69	NaN
moesi5	$1.35 \cdot 10^{5}$	246.02	0.13	0.68	NaN
multi-me	60,292	6.83	$6.3 \cdot 10^{-2}$	$2.79 \cdot 10^{-2}$	NaN
multipoll	59,696	11.79	$3.52 \cdot 10^{-2}$	$6.65 \cdot 10^{-2}$	NaN
newdekker	$1.68 \cdot 10^{5}$	64.25	2.81	2.71	NaN
newrtp	60,683	5.21	$2.99 \cdot 10^{-2}$	$2.13 \cdot 10^{-2}$	NaN
peterson	60,157	362.37	$9.24 \cdot 10^{-2}$	1.48	NaN
pncsacover	$1.21 \cdot 10^{5}$	141.27	5.76	1.43	38
pncsasemiliv	$1.1 \cdot 10^{5}$	798.43	0.22	3.26	10
queuedbusyflag	44	26,596.9	20,318.75	2,411.76	NaN
read-write	$1.11 \cdot 10^{5}$	752.01	0.9	2.45	NaN
simplejavaexample	$1.28 \cdot 10^{5}$	98.28	4.3	1.1	13
transthesis	14,395	67.98	45.91	2.53	NaN

Table C.6.: Benchmark data for AGI in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	60,601	3.46	$5.96 \cdot 10^{-2}$	$1.41 \cdot 10^{-2}$	NaN
basicextransfer	$2.32 \cdot 10^{5}$	250.16	$9.5 \cdot 10^{-2}$	0.52	NaN
bingham-h250	135	7,399.57	2,560.04	2,305.92	NaN
consprod	$2.13 \cdot 10^{5}$	183.41	2.05	1.37	NaN
consprod2	59,764	1,144.88	$6.07 \cdot 10^{-2}$	4.78	NaN
csm	56,868	40.69	$2.84 \cdot 10^{-2}$	2.04	NaN
csm-broad	$2.77 \cdot 10^{5}$	220.44	1.18	0.83	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	240	4,737.87	3,734.98	315.6	17
delegatebuffer.16.1	76	17,120.86	11,826.04	1,424.84	18
efm	$4.04 \cdot 10^{5}$	634.02	0.25	1.15	NaN
examplelea	63	43,028.9	8,055.04	13,464.02	NaN
ext-rw	59,601	720.66	$6.78 \cdot 10^{-2}$	2.95	NaN
ext-rw-smallconsts	59,340	8.71	$6.76 \cdot 10^{-2}$	$6.03 \cdot 10^{-2}$	NaN
fms	$1.16 \cdot 10^{5}$	11.22	$3 \cdot 10^{-2}$	$6.41 \cdot 10^{-2}$	NaN
fms2	$1.16 \cdot 10^{5}$	13.61	$7.41 \cdot 10^{-2}$	$7.98 \cdot 10^{-2}$	NaN
german	$1.98 \cdot 10^{5}$	159.61	0.1	1.24	NaN
glotter	20,204	219.07	33.17	11.6	7
hts	6	$1.47 \cdot 10^{5}$	$1.32 \cdot 10^{5}$	7,635.76	29
java	1,906	792.64	467.07	46.34	15
java.10.0	1,498	887.06	594.53	64.64	14
java.11.0	3,938	411.95	222.86	19.23	15
java2	188	7,047.03	4,742.84	604.03	NaN
java2.10.2	1,410	1,144.19	630.86	57.18	14
java2.11.2	1,171	1,131.64	763.45	70.71	15
kanban	$1.02 \cdot 10^{5}$	25.92	8.38	0.58	88
km-nonterm.4.3	$2.17 \cdot 10^{5}$	1,590.35	0.25	3.44	18
km-nonterm.5.4	$3.96 \cdot 10^{5}$	274	0.28	0.76	28
km-nonterm.6.5	$3.76 \cdot 10^{5}$	94.22	0.38	0.4	40
lamport	60,332	6.73	$5.23 \cdot 10^{-2}$	$3.24 \cdot 10^{-2}$	NaN
leabasicapproach	59,870	905.89	0.17	4.06	5
leaconflictset	32,404	51.34	26.23	1.73	20
lifo	$2.24 \cdot 10^{5}$	968.17	$6.11 \cdot 10^{-2}$	2.11	NaN
manufacturing	60,438	2,824.63	$4.92 \cdot 10^{-2}$	11.49	NaN
mesh2x2	$1.13 \cdot 10^{5}$	14.03	$4.57 \cdot 10^{-2}$	$9.95 \cdot 10^{-2}$	NaN
mesh3x2	$1.58 \cdot 10^{5}$	20.22	$5.99 \cdot 10^{-2}$	0.17	NaN
moesi	$1.18 \cdot 10^{5}$	234.68	$4.41 \cdot 10^{-2}$	0.69	NaN
moesi5	$1.38 \cdot 10^{5}$	232.72	0.12	0.65	NaN
multi-me	60,469	12.29	$6.11 \cdot 10^{-2}$	$5 \cdot 10^{-2}$	NaN
multipoll	59,934	10.93	$3.68 \cdot 10^{-2}$	$6.35 \cdot 10^{-2}$	NaN
newdekker	$1.72 \cdot 10^{5}$	135.44	2.7	2.73	NaN
newrtp	$1.04 \cdot 10^{5}$	202.28	$3.26 \cdot 10^{-2}$	5.9	NaN
peterson	60,034	972.41	$9.08 \cdot 10^{-2}$	4.71	NaN
pncsacover	97,701	191.95	7.53	1.58	38
pncsasemiliv	$1.09 \cdot 10^{5}$	975.45	0.23	2.96	10
queuedbusyflag	44	27,321.24	19,818.77	2,777.1	NaN
read-write	$1.11 \cdot 10^{5}$	544.21	0.9	1.92	NaN
simplejavaexample	$1.3 \cdot 10^{5}$	110.05	4.24	1.16	13
transthesis	13,820	77.31	47.64	4.21	NaN

Table C.7.: Benchmark data for GI in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	$5.04 \cdot 10^6$	2,130.6	0.1	2.34	NaN
basicextransfer	$7.33 \cdot 10^6$	8,197.24	$7.52 \cdot 10^{-2}$	3.62	NaN
bingham-h250	107	11,805.81	9,953.22	2,555.99	NaN
consprod	249	5,675.64	3,566.26	520.79	NaN
consprod2	50,872	40.02	17.18	1.43	NaN
csm	$3.46 \cdot 10^6$	6,373.33	0.17	5.36	NaN
csm-broad	$7 \cdot 10^5$	32.65	1.16	0.32	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	164	7,691.4	5,428.85	490.31	17
delegatebuffer.16.1	57	22,614.77	15,856.88	1,948.71	18
efm	$3.14 \cdot 10^6$	3,794.5	0.21	2.82	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	67	18,677.17	12,967.4	2,611.59	NaN
fms	$5.18 \cdot 10^6$	5,382.04	$5.37 \cdot 10^{-2}$	5.49	NaN
fms2	$9.91 \cdot 10^5$	972.47	0.75	3.58	NaN
german	$3.62 \cdot 10^5$	17.48	2.21	0.56	NaN
glotter	4,483	312.76	198.73	15.16	7
hts	84	13,369.21	10,706.65	1,038.75	29
java	344	3,917.42	2,589.92	280.33	15
java.10.0	1,129	1,130.45	794.22	64.72	14
java.11.0	2,740	555.24	323.03	30.92	15
java2	19	68,003.52	48,092.05	7,830.72	NaN
java2.10.2	937	1,326.75	953.77	82.16	14
java2.11.2	788	1,609.91	1,132.06	94.62	15
kanban	$1.46 \cdot 10^6$	3,974.07	0.44	5.85	109
km-nonterm.4.3	$3.09 \cdot 10^6$	6,551.47	0.2	4.16	18
km-nonterm.5.4	$2.43 \cdot 10^6$	2,086.01	0.27	1.51	28
km-nonterm.6.5	$1.8 \cdot 10^6$	360.4	0.38	0.4	40
lamport	$9.28 \cdot 10^5$	49.73	0.8	0.54	NaN
leabasicapproach	$5.33 \cdot 10^6$	9,860.43	$6.73 \cdot 10^{-2}$	6.03	5
leaconflictset	8,454	215.11	105.11	9.39	20
lifo	$7.37 \cdot 10^6$	4,462.35	$2.2 \cdot 10^{-2}$	2.46	NaN
manufacturing	1,212	989.73	745.29	69.07	NaN
mesh2x2	$2.78 \cdot 10^6$	5,957.9	0.14	9	NaN
mesh3x2	$1.76 \cdot 10^6$	5,815.19	0.18	13.58	NaN
moesi	$7.74 \cdot 10^6$	5,397.55	$1.77 \cdot 10^{-2}$	2.79	NaN
moesi5	$1.6 \cdot 10^6$	5,422.95	$7.16 \cdot 10^{-2}$	5.43	NaN
multi-me	$1.08 \cdot 10^6$	2,275.84	0.73	2.87	NaN
multipoll	$2.01 \cdot 10^6$	2,858.27	0.31	6.02	NaN
newdekker	65,061	38.92	13.47	1.2	NaN
newrtp	$4.97 \cdot 10^6$	7,318.98	0.12	5.36	NaN
peterson	$1.31 \cdot 10^5$	24.6	6.63	0.7	NaN
pncsacover	$2.38 \cdot 10^5$	39.61	3.48	0.69	35
pncsasemiliv	$2.94 \cdot 10^6$	5,145.8	0.11	10.18	10
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.35 \cdot 10^5$	28.04	6.44	0.77	NaN
simplejavaexample	30,918	52.83	28.46	2.14	11
transthesis	8	$1.38 \cdot 10^5$	$1.21 \cdot 10^5$	10,141.55	NaN

Table C.8.: Benchmark data for AGP in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	$5.57 \cdot 10^6$	2,131.19	0.11	2.62	NaN
basicextransfer	$5.84 \cdot 10^6$	3,033.61	$9.72 \cdot 10^{-2}$	3.28	NaN
bingham-h250	113	12,048.94	6,414.91	2,493.74	NaN
consprod	256	5,474.29	3,372.46	538.03	NaN
consprod2	43,530	46.89	20.17	1.35	NaN
csm	$3.43 \cdot 10^6$	2,863.38	0.17	3.9	NaN
csm-broad	$7.02 \cdot 10^5$	39.71	1.15	0.33	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	163	7,667.78	5,531.4	466.5	17
delegatebuffer.16.1	59	21,570.79	15,025.99	1,805.9	18
efm	$3.18 \cdot 10^6$	1,908.08	0.21	2.54	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	65	18,528.57	13,465.04	2,364	NaN
fms	$5.22 \cdot 10^6$	6,849.37	$5.43 \cdot 10^{-2}$	7.74	NaN
fms2	$9.31 \cdot 10^5$	942.28	0.79	3.79	NaN
german	$3.73 \cdot 10^5$	23.74	2.25	0.34	NaN
glotter	4,476	339.58	199.32	15.51	7
hts	81	14,074.79	11,211.23	955.52	29
java	351	3,787.34	2,536.72	272.66	15
java.10.0	1,116	1,126.54	801.32	69.39	14
java.11.0	2,705	543.49	327.45	30.42	15
java2	20	65,901.85	44,822.31	6,826.3	NaN
java2.10.2	884	1,491.22	1,011.03	98.78	14
java2.11.2	790	1,677.9	1,124.99	99.24	15
kanban	$3.6 \cdot 10^5$	33.48	2.21	0.52	81
km-nonterm.4.3	$3.22 \cdot 10^6$	2,713.63	0.19	1.98	18
km-nonterm.5.4	$2.39 \cdot 10^6$	1,430.19	0.27	1.09	28
km-nonterm.6.5	$1.81 \cdot 10^6$	311.93	0.38	0.33	40
lamport	$1.02 \cdot 10^6$	387.34	0.78	0.65	NaN
leabasicapproach	$5.19 \cdot 10^6$	4,943.65	$6.8 \cdot 10^{-2}$	5.05	5
leaconflictset	7,648	207.22	116.4	10.91	20
lifo	$7.52 \cdot 10^6$	6,453.33	$2.4 \cdot 10^{-2}$	3.85	NaN
manufacturing	1,447	881.43	621.22	61.73	NaN
mesh2x2	$2.68 \cdot 10^6$	5,170.81	0.15	10.05	NaN
mesh3x2	$1.89 \cdot 10^6$	5,853.88	0.17	8.18	NaN
moesi	$7.9 \cdot 10^6$	4,772.07	$1.61 \cdot 10^{-2}$	1.92	NaN
moesi5	$1.66 \cdot 10^6$	3,954	$6.98 \cdot 10^{-2}$	3.44	NaN
multi-me	$1.15 \cdot 10^6$	2,188.62	0.68	2.74	NaN
multipoll	$2.07 \cdot 10^6$	2,076.16	0.3	5.87	NaN
newdekker	64,561	35.13	13.57	1.22	NaN
newrtp	$5.05 \cdot 10^6$	7,977.93	0.12	5.94	NaN
peterson	$1.31 \cdot 10^5$	24.08	6.64	0.72	NaN
pncsacover	$2.41 \cdot 10^5$	55.23	3.43	0.75	35
pncsasemiliv	$2.77 \cdot 10^6$	6,319.69	0.13	9.5	10
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.37 \cdot 10^5$	25.93	6.37	0.76	NaN
simplejavaexample	31,089	51.6	28.27	2.17	11
transthesis	8	$1.52 \cdot 10^5$	$1.24 \cdot 10^5$	15,313.05	NaN

Table C.9.: Benchmark data for GP in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	$5.06 \cdot 10^6$	5,136.75	0.1	3.28	NaN
basicextransfer	$7.24 \cdot 10^6$	2,143.27	$7.7 \cdot 10^{-2}$	2.04	NaN
bingham-h250	85	14,982.59	12,105.28	2,630.79	NaN
consprod	252	5,481.98	3,452.37	525.34	NaN
consprod2	50,164	37.92	17.46	1.34	NaN
csm	$2.65 \cdot 10^6$	4,974.88	0.25	4.66	NaN
csm-broad	$7.14 \cdot 10^5$	108.39	1.14	0.62	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	159	7,853.59	5,595.34	528.84	17
delegatebuffer.16.1	58	19,596.41	15,195.49	1,744.2	18
efm	$3.08 \cdot 10^6$	5,007.63	0.21	3.63	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	30	42,520.48	30,468.86	5,228.85	NaN
fms	$4.82 \cdot 10^6$	7,068.16	$6.5 \cdot 10^{-2}$	7.43	NaN
fms2	70,671	34.6	12.2	1.48	NaN
german	$3.79 \cdot 10^5$	44.97	2.21	0.4	NaN
glotter	4,118	338.61	216.73	17.54	7
hts	9	$1.23 \cdot 10^5$	$1.11 \cdot 10^5$	9,771.3	29
java	354	3,794.99	2,519.02	257.26	15
java.10.0	1,008	1,276.27	884.64	80.94	14
java.11.0	2,717	635.05	325.4	31.99	15
java2	18	74,757.25	51,032.64	10,596.97	NaN
java2.10.2	925	1,510.94	962.73	80.22	14
java2.11.2	791	1,518.78	1,123.57	92.94	15
kanban	$3.57 \cdot 10^5$	27.84	2.27	0.44	110
km-nonterm.4.3	$3.2 \cdot 10^6$	1,033.91	0.19	1.28	18
km-nonterm.5.4	$2.33 \cdot 10^6$	579.61	0.29	0.68	28
km-nonterm.6.5	$1.45 \cdot 10^6$	347.4	0.47	0.47	40
lamport	$7.35 \cdot 10^5$	29.61	1.02	0.56	NaN
leabasicapproach	$4.99 \cdot 10^6$	3,404.73	$7.99 \cdot 10^{-2}$	3.6	5
leaconflictset	8,250	291.83	107.5	11.27	20
lifo	$7.5 \cdot 10^6$	6,762.54	$2.34 \cdot 10^{-2}$	4.52	NaN
manufacturing	1,476	931.53	609.69	64.75	NaN
mesh2x2	$1.25 \cdot 10^6$	2,102.78	0.52	6.49	NaN
mesh3x2	$8.94 \cdot 10^5$	2,560.54	0.67	9.96	NaN
moesi	$1.75 \cdot 10^6$	2,174.75	$1.61 \cdot 10^{-2}$	4.14	NaN
moesi5	$1.61 \cdot 10^6$	5,319.89	$6.94 \cdot 10^{-2}$	8.59	NaN
multi-me	$1.02 \cdot 10^6$	3,078.62	0.78	3.59	NaN
multipoll	55,726	34.08	15.71	1.04	NaN
newdekker	65,231	37.34	13.44	1.15	NaN
newrtp	$3 \cdot 10^6$	5,357.09	0.23	4.5	NaN
peterson	$1.34 \cdot 10^5$	33.44	6.47	0.69	NaN
pncsacover	26,302	54.59	33.58	2.08	38
pncsasemiliv	$2.7 \cdot 10^6$	5,711.22	0.14	7.83	10
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.05 \cdot 10^5$	25.63	8.32	0.97	NaN
simplejavaexample	27,487	60.95	32.06	2.22	11
transthesis	7	$1.81 \cdot 10^5$	$1.45 \cdot 10^5$	17,845.05	NaN

Table C.10.: Benchmark data for AG in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	$5.36 \cdot 10^6$	1,774.03	0.11	2.07	NaN
basicextransfer	$7.18 \cdot 10^6$	8,844.64	$7.55 \cdot 10^{-2}$	3.85	NaN
bingham-h250	81	14,934.65	12,527.25	2,826.91	NaN
consprod	266	5,337	3,303.37	489.61	NaN
consprod2	50,623	50.45	17.3	1.4	NaN
csm	$2.77 \cdot 10^6$	6,280.61	0.23	5.24	NaN
csm-broad	$6.66 \cdot 10^5$	33.36	1.22	0.33	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	167	7,547.13	5,309.15	513.29	17
delegatebuffer.16.1	58	20,196.8	15,524.54	1,669.49	18
efm	$3.26 \cdot 10^6$	4,253.04	0.2	3.16	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	29	40,458.15	31,163.17	4,160.75	NaN
fms	$4.44 \cdot 10^6$	17,045.53	$6.86 \cdot 10^{-2}$	13.34	NaN
fms2	68,897	34.74	12.54	1.5	NaN
german	$3.68 \cdot 10^5$	17.3	2.29	0.37	NaN
glotter	3,877	342.94	227.92	22.48	7
hts	9	$1.2 \cdot 10^5$	$1.07 \cdot 10^5$	9,296.21	29
java	349	4,442.49	2,544.63	301.64	15
java.10.0	1,118	1,140.01	797.83	66.2	14
java.11.0	2,694	562.33	328.8	30.71	15
java2	18	76,057.5	49,058.83	9,502.74	NaN
java2.10.2	945	1,322.93	947.98	80.33	14
java2.11.2	814	1,505.32	1,096.98	92.85	15
kanban	$1.04 \cdot 10^5$	37.47	8.35	0.65	88
km-nonterm.4.3	$2.95 \cdot 10^6$	3,361.71	0.21	2.54	18
km-nonterm.5.4	$2.43 \cdot 10^6$	591.68	0.27	0.64	28
km-nonterm.6.5	$1.86 \cdot 10^6$	336.6	0.37	0.51	40
lamport	$7.98 \cdot 10^5$	24.85	1	0.34	NaN
leabasicapproach	$5.12 \cdot 10^6$	3,358.93	$7.79 \cdot 10^{-2}$	4.51	5
leaconflictset	8,424	175.58	105.5	9.49	20
lifo	$7.41 \cdot 10^6$	6,638.17	$2.47 \cdot 10^{-2}$	2.78	NaN
manufacturing	1,459	1,056.45	615.76	59.86	NaN
mesh2x2	$1.21 \cdot 10^6$	7,872.34	0.53	11.23	NaN
mesh3x2	$9.13 \cdot 10^5$	3,995.66	0.64	13.93	NaN
moesi	$7.63 \cdot 10^6$	6,222.43	$1.58 \cdot 10^{-2}$	3.1	NaN
moesi5	$1.56 \cdot 10^6$	4,992.37	$7.07 \cdot 10^{-2}$	7.21	NaN
multi-me	$1 \cdot 10^6$	510.76	0.79	1.91	NaN
multipoll	54,165	31.38	16.06	1.32	NaN
newdekker	64,150	31.93	13.67	1.17	NaN
newrtp	$2.92 \cdot 10^6$	3,675.58	0.23	4.1	NaN
peterson	$1.29 \cdot 10^5$	26.48	6.7	0.73	NaN
pncsacover	26,551	55.87	33.23	2.16	38
pncsasemiliv	$2.67 \cdot 10^6$	5,846.31	0.14	9.36	10
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.03 \cdot 10^5$	23.01	8.49	0.99	NaN
simplejavaexample	31,336	58.29	28.05	2.16	11
transthesis	7	$1.72 \cdot 10^5$	$1.4 \cdot 10^5$	15,774.4	NaN

Table C.11.: Benchmark data for G in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	60,826	0.2	$6.21 \cdot 10^{-2}$	$3.04 \cdot 10^{-3}$	NaN
basicextransfer	$2.27 \cdot 10^5$	596.54	0.12	1.36	NaN
bingham-h250	131	7,460.39	6,697.92	2,338.61	NaN
consprod	$1.94 \cdot 10^5$	120.4	2.46	1.03	NaN
consprod2	59,958	1,048.09	$6.04 \cdot 10^{-2}$	4.38	NaN
csm	60,318	10.06	$2.65 \cdot 10^{-2}$	$5.36 \cdot 10^{-2}$	NaN
csm-broad	91,541	331.58	7.46	8.27	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$3.76 \cdot 10^5$	556.76	0.4	1.13	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	59,411	11.37	$7.11 \cdot 10^{-2}$	$6.57 \cdot 10^{-2}$	NaN
ext-rw-smallconsts	59,614	763.2	$6.94 \cdot 10^{-2}$	3.13	NaN
fms	$1.17 \cdot 10^5$	13.69	$2.99 \cdot 10^{-2}$	$7.49 \cdot 10^{-2}$	NaN
fms2	$2.2 \cdot 10^5$	230.87	$7.24 \cdot 10^{-2}$	0.7	NaN
german	32	17.31	0.38	3.57	NaN
glotter	21	14,494.85	1,074.61	2,897.87	11
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	70,177	2,947.56	1.71	65.65	146
km-nonterm.4.3	$4.12 \cdot 10^5$	1,068.37	0.21	1.8	25
km-nonterm.5.4	$3.96 \cdot 10^5$	257.69	0.3	0.87	37
km-nonterm.6.5	$3.8 \cdot 10^5$	224.87	0.37	0.61	51
lamport	60,434	213.82	$5.49 \cdot 10^{-2}$	0.87	NaN
leabasicapproach	$1.15 \cdot 10^5$	1,934.93	0.24	5.87	11
leaconflictset	3	53,805.09	48,324.68	19,880.32	891
lifo	$2.24 \cdot 10^5$	1,088.78	$6.4 \cdot 10^{-2}$	2.3	NaN
manufacturing	60,742	4.58	$4.72 \cdot 10^{-2}$	$1.86 \cdot 10^{-2}$	NaN
mesh2x2	$1.12 \cdot 10^5$	8.25	$4.17 \cdot 10^{-2}$	$7.2 \cdot 10^{-2}$	NaN
mesh3x2	$1.61 \cdot 10^5$	15.55	$6.1 \cdot 10^{-2}$	0.13	NaN
moesi	$1.17 \cdot 10^5$	232.04	$4.52 \cdot 10^{-2}$	0.68	NaN
moesi5	$1.35 \cdot 10^5$	678.79	0.12	1.96	NaN
multi-me	60,284	5.83	$6.2 \cdot 10^{-2}$	$2.87 \cdot 10^{-2}$	NaN
multipoll	$1.17 \cdot 10^5$	11.55	$3.62 \cdot 10^{-2}$	$6.58 \cdot 10^{-2}$	NaN
newdekker	728	$2.36 \cdot 10^5$	19.34	9,737.79	NaN
newrtp	60,549	7.05	$2.85 \cdot 10^{-2}$	$2.86 \cdot 10^{-2}$	NaN
peterson	59,993	6.76	$8.39 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	NaN
pncsacover	1	282.4	282.4	0	2,410
pncsasemiliv	5	$3.35 \cdot 10^5$	4,752.91	$1.33 \cdot 10^5$	569
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.82 \cdot 10^5$	566.18	0.87	2.62	NaN
simplejavaexample	3	$1.29 \cdot 10^5$	96,497.48	43,371.6	171
transthesis	12,667	84.76	54.61	2.7	NaN

Table C.12.: Benchmark data for AIP in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	60,720	3.72	$5.91 \cdot 10^{-2}$	$1.52 \cdot 10^{-2}$	NaN
basicextransfer	$2.27 \cdot 10^{5}$	278.99	0.12	0.59	NaN
bingham-h250	136	7,385.07	2,646.99	2,213.42	NaN
consprod	$1.95 \cdot 10^{5}$	345.42	2.41	1.44	NaN
consprod2	59,998	230.91	$5.98 \cdot 10^{-2}$	0.95	NaN
csm	60,114	1,920.6	$2.55 \cdot 10^{-2}$	7.83	NaN
csm-broad	90,673	482.32	7.54	8.54	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$3.74 \cdot 10^{5}$	173.7	0.4	0.74	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	59,741	1,018.68	$6.6 \cdot 10^{-2}$	4.17	NaN
ext-rw-smallconsts	59,585	10.38	$6.79 \cdot 10^{-2}$	$6.7 \cdot 10^{-2}$	NaN
fms	$1.17 \cdot 10^{5}$	14.01	$3.03 \cdot 10^{-2}$	$8.13 \cdot 10^{-2}$	NaN
fms2	$2.2 \cdot 10^{5}$	479.79	$7.66 \cdot 10^{-2}$	1.41	NaN
german	12	15.65	0.61	4.88	NaN
glotter	4	2,199.95	800.3	655.15	11
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	35,863	3,104.83	5.56	89.37	168
km-nonterm.4.3	$4.02 \cdot 10^{5}$	506.99	0.26	1.3	25
km-nonterm.5.4	$3.95 \cdot 10^{5}$	798.86	0.29	1.45	37
km-nonterm.6.5	$3.78 \cdot 10^{5}$	244.19	0.39	0.62	51
lamport	60,494	5.85	$5.54 \cdot 10^{-2}$	$2.98 \cdot 10^{-2}$	NaN
leabasicapproach	$1.14 \cdot 10^{5}$	5,280.01	0.24	15.73	11
leaconflictset	1	15,351.37	15,351.37	0	196
lifo	$2.23 \cdot 10^{5}$	238.2	$6.41 \cdot 10^{-2}$	0.51	NaN
manufacturing	61,006	2.57	$3.9 \cdot 10^{-2}$	$1.08 \cdot 10^{-2}$	NaN
mesh2x2	$1.12 \cdot 10^{5}$	11.64	$4.38 \cdot 10^{-2}$	$8.5 \cdot 10^{-2}$	NaN
mesh3x2	$1.6 \cdot 10^{5}$	478.01	$5.69 \cdot 10^{-2}$	1.21	NaN
moesi	$1.17 \cdot 10^{5}$	230.88	$4.78 \cdot 10^{-2}$	0.68	NaN
moesi5	$1.27 \cdot 10^{5}$	616.51	0.13	5.61	NaN
multi-me	60,428	7.21	$6.1 \cdot 10^{-2}$	$3.4 \cdot 10^{-2}$	NaN
multipoll	59,953	15.82	$3.68 \cdot 10^{-2}$	$9.43 \cdot 10^{-2}$	NaN
newdekker	458	$1.63 \cdot 10^{5}$	21.58	8,552.6	NaN
newrtp	60,396	210.82	$2.91 \cdot 10^{-2}$	0.86	NaN
peterson	60,087	1,714.63	$8.55 \cdot 10^{-2}$	6.99	NaN
pncsacover	1	3,068.49	3,068.49	0	5,673
pncsasemiliv	7	71,247.81	1,137.44	24,447.02	529
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.78 \cdot 10^{5}$	354.74	0.87	2.65	NaN
simplejavaexample	2	11,374.31	10,176.69	1,197.62	95.5
transthesis	12,740	79.62	54.19	3.3	NaN

Table C.13.: Benchmark data for IP in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	60,733	0.27	$5.97 \cdot 10^{-2}$	$2.68 \cdot 10^{-3}$	NaN
basicextransfer	$2.27 \cdot 10^{5}$	572.32	0.12	1.51	NaN
bingham-h250	129	7,947.19	2,757.4	2,384.47	NaN
consprod	$1.95 \cdot 10^{5}$	350.02	2.42	1.38	NaN
consprod2	59,888	237	$6.14 \cdot 10^{-2}$	0.97	NaN
csm	60,482	7.68	$2.54 \cdot 10^{-2}$	$3.12 \cdot 10^{-2}$	NaN
csm-broad	88,498	295.43	7.71	8.66	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$3.67 \cdot 10^{5}$	183	0.44	0.74	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	59,542	10.68	$6.69 \cdot 10^{-2}$	$6.25 \cdot 10^{-2}$	NaN
ext-rw-smallconsts	59,593	636.63	$6.95 \cdot 10^{-2}$	2.61	NaN
fms	$1.17 \cdot 10^{5}$	11.56	$2.87 \cdot 10^{-2}$	$6.51 \cdot 10^{-2}$	NaN
fms2	$1.16 \cdot 10^{5}$	658.77	$7.6 \cdot 10^{-2}$	1.94	NaN
german	17	14.52	0.33	4.57	NaN
glotter	44	$2.13 \cdot 10^{5}$	878.13	35,298.55	11
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	2	217.28	134.79	82.49	1,034
km-nonterm.4.3	$4.07 \cdot 10^{5}$	425.51	0.21	1.09	25
km-nonterm.5.4	$3.97 \cdot 10^{5}$	351.28	0.29	0.69	37
km-nonterm.6.5	$3.73 \cdot 10^{5}$	233.39	0.41	0.63	51
lamport	60,699	8.07	$4.99 \cdot 10^{-2}$	$3.27 \cdot 10^{-2}$	NaN
leabasicapproach	$2 \cdot 10^{5}$	3,034.66	0.32	10.05	17
leaconflictset	0	NaN	NaN	NaN	NaN
lifo	$2.26 \cdot 10^{5}$	14.86	$5.83 \cdot 10^{-2}$	$6.46 \cdot 10^{-2}$	NaN
manufacturing	60,644	2.67	$4.91 \cdot 10^{-2}$	$1.12 \cdot 10^{-2}$	NaN
mesh2x2	$1.12 \cdot 10^{5}$	12.36	$4.55 \cdot 10^{-2}$	$8.24 \cdot 10^{-2}$	NaN
mesh3x2	$1.59 \cdot 10^{5}$	21.21	$5.9 \cdot 10^{-2}$	0.17	NaN
moesi	$1.17 \cdot 10^{5}$	225.3	$4.52 \cdot 10^{-2}$	0.66	NaN
moesi5	$1.37 \cdot 10^{5}$	487.47	0.12	1.47	NaN
multi-me	60,232	694.63	$6.07 \cdot 10^{-2}$	2.83	NaN
multipoll	$1.16 \cdot 10^{5}$	471.93	$3.74 \cdot 10^{-2}$	1.39	NaN
newdekker	1,575	$1.29 \cdot 10^{5}$	21.63	5,104.23	NaN
newrtp	60,572	7.76	$2.87 \cdot 10^{-2}$	$3.16 \cdot 10^{-2}$	NaN
peterson	60,157	1,051.81	$8.79 \cdot 10^{-2}$	4.4	NaN
pncsacover	1	$1.76 \cdot 10^{5}$	$1.76 \cdot 10^{5}$	0	$4.36 \cdot 10^{5}$
pncsasemiliv	0	NaN	NaN	NaN	NaN
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.67 \cdot 10^{5}$	380.17	0.97	2.76	NaN
simplejavaexample	1	5,151.58	5,151.58	0	108
transthesis	12,638	102.95	54.65	3.14	NaN

Table C.14.: Benchmark data for AI in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	60,608	5.22	$6.14 \cdot 10^{-2}$	$2.12 \cdot 10^{-2}$	NaN
basicextransfer	$2.27 \cdot 10^{5}$	664.75	0.12	1.4	NaN
bingham-h250	129	7,892.83	4,732.44	2,403.49	NaN
consprod	$1.89 \cdot 10^{5}$	196.5	2.59	1.3	NaN
consprod2	59,818	6.74	$6.1 \cdot 10^{-2}$	$4.22 \cdot 10^{-2}$	NaN
csm	60,654	9.73	$2.44 \cdot 10^{-2}$	$5.21 \cdot 10^{-2}$	NaN
csm-broad	89,764	356.88	7.58	8.95	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$3.73 \cdot 10^{5}$	196.81	0.41	0.95	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	59,584	785.04	$6.97 \cdot 10^{-2}$	3.35	NaN
ext-rw-smallconsts	59,541	615.41	$6.77 \cdot 10^{-2}$	2.52	NaN
fms	$1.18 \cdot 10^{5}$	10.02	$2.74 \cdot 10^{-2}$	$6.12 \cdot 10^{-2}$	NaN
fms2	$2.2 \cdot 10^{5}$	485.94	$7.35 \cdot 10^{-2}$	1.04	NaN
german	25	22.32	0.46	6.62	NaN
glotter	14	1,874.75	771.52	413.68	11
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	1	7.79	7.79	0	300
km-nonterm.4.3	$4.13 \cdot 10^{5}$	316.8	0.21	0.85	25
km-nonterm.5.4	$3.96 \cdot 10^{5}$	908.29	0.29	1.57	37
km-nonterm.6.5	$3.74 \cdot 10^{5}$	846.06	0.39	1.63	51
lamport	60,582	5.69	$5.15 \cdot 10^{-2}$	$2.45 \cdot 10^{-2}$	NaN
leabasicapproach	$1.06 \cdot 10^{5}$	33,487.17	0.32	104.29	17
leaconflictset	6	99,130.4	15,661.27	32,932.1	567
lifo	$2.25 \cdot 10^{5}$	241.31	$5.97 \cdot 10^{-2}$	0.51	NaN
manufacturing	60,503	4.55	$4.76 \cdot 10^{-2}$	$2.17 \cdot 10^{-2}$	NaN
mesh2x2	$1.13 \cdot 10^{5}$	12.57	$4.47 \cdot 10^{-2}$	$8.98 \cdot 10^{-2}$	NaN
mesh3x2	$1.57 \cdot 10^{5}$	15.97	$5.85 \cdot 10^{-2}$	0.13	NaN
moesi	$1.18 \cdot 10^{5}$	448.12	$4.29 \cdot 10^{-2}$	1.31	NaN
moesi5	$1.37 \cdot 10^{5}$	536.82	0.12	2.09	NaN
multi-me	60,413	3,048.4	$6.24 \cdot 10^{-2}$	12.4	NaN
multipoll	59,912	10.64	$3.54 \cdot 10^{-2}$	$6.41 \cdot 10^{-2}$	NaN
newdekker	767	$2.2 \cdot 10^{5}$	21.58	9,945.97	NaN
newrtp	60,516	8.95	$2.82 \cdot 10^{-2}$	$4.44 \cdot 10^{-2}$	NaN
peterson	60,088	1,678.05	$8.66 \cdot 10^{-2}$	6.85	NaN
pncsacover	0	NaN	NaN	NaN	NaN
pncsasemiliv	3	$1.39 \cdot 10^{5}$	69,983.42	65,567.84	37,983
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	$1.68 \cdot 10^{5}$	406.51	0.95	2.72	NaN
simplejavaexample	0	NaN	NaN	NaN	NaN
transthesis	12,663	84.17	54.24	2.65	NaN

Table C.15.: Benchmark data for I in Sect. 7.2.3

Model	**Iter.**	**Max. ms**	**Med. ms**	**σ ms**	**Med. T**
basic-me	$4.15 \cdot 10^6$	3,147.68	0.16	2.37	NaN
basicextransfer	$7.09 \cdot 10^6$	2,626.39	$7.35 \cdot 10^{-2}$	2.35	NaN
bingham-h250	96	16,840.26	10,034.68	3,453.56	NaN
consprod	0	NaN	NaN	NaN	NaN
consprod2	0	NaN	NaN	NaN	NaN
csm	14	$1.31 \cdot 10^5$	28.51	33,659.82	NaN
csm-broad	634	35,604.33	21.74	1,755.67	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$1.06 \cdot 10^6$	405.5	0.64	0.71	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	0	NaN	NaN	NaN	NaN
fms	$1.35 \cdot 10^6$	4,018.98	0.16	5.51	NaN
fms2	73,495	11,183.07	4.9	54.51	NaN
german	1	3,294.36	3,294.36	0	NaN
glotter	0	NaN	NaN	NaN	NaN
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	65,927	2,547.77	1.81	71.44	146
km-nonterm.4.3	$2.61 \cdot 10^6$	1,943.86	0.25	1.51	25
km-nonterm.5.4	$2.01 \cdot 10^6$	1,101.2	0.34	1.82	37
km-nonterm.6.5	$1.47 \cdot 10^6$	302.27	0.49	0.32	51
lamport	1	239.59	239.59	0	NaN
leabasicapproach	3	$4.09 \cdot 10^5$	$2.19 \cdot 10^5$	$1.86 \cdot 10^5$	10
leaconflictset	0	NaN	NaN	NaN	NaN
lifo	$7.45 \cdot 10^6$	4,575.62	$2.25 \cdot 10^{-2}$	2.64	NaN
manufacturing	0	NaN	NaN	NaN	NaN
mesh2x2	$1.52 \cdot 10^6$	6,037.23	0.35	7.42	NaN
mesh3x2	$6.79 \cdot 10^5$	1,655.3	0.7	8.33	NaN
moesi	$7.56 \cdot 10^6$	4,988.73	$1.62 \cdot 10^{-2}$	2.21	NaN
moesi5	$1.57 \cdot 10^6$	3,797.74	$6.59 \cdot 10^{-2}$	4.03	NaN
multi-me	170	$4.91 \cdot 10^5$	63.7	45,033.75	NaN
multipoll	$1.42 \cdot 10^6$	4,630.65	0.5	6.43	NaN
newdekker	0	NaN	NaN	NaN	NaN
newrtp	$5.45 \cdot 10^6$	3,175.15	0.1	4.43	NaN
peterson	0	NaN	NaN	NaN	NaN
pncsacover	0	NaN	NaN	NaN	NaN
pncsasemiliv	0	NaN	NaN	NaN	NaN
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	0	NaN	NaN	NaN	NaN
simplejavaexample	0	NaN	NaN	NaN	NaN
transthesis	0	NaN	NaN	NaN	NaN

Table C.16.: Benchmark data for AP in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	$4.15 \cdot 10^6$	3,457.83	0.16	2.53	NaN
basicextransfer	$6.92 \cdot 10^6$	11,160.35	$7.43 \cdot 10^{-2}$	4.9	NaN
bingham-h250	90	16,983.35	10,221.42	3,331.12	NaN
consprod	0	NaN	NaN	NaN	NaN
consprod2	0	NaN	NaN	NaN	NaN
csm	24	$1.22 \cdot 10^5$	22.83	24,386.92	NaN
csm-broad	519	$1.36 \cdot 10^5$	22.36	6,375.55	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$1.03 \cdot 10^6$	364.42	0.67	0.68	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	0	NaN	NaN	NaN	NaN
fms	$1.29 \cdot 10^6$	3,920.43	0.16	5.94	NaN
fms2	76,802	4,037.73	4.68	45.36	NaN
german	0	NaN	NaN	NaN	NaN
glotter	0	NaN	NaN	NaN	NaN
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	36,211	2,496.4	5.49	82.03	167
km-nonterm.4.3	$2.64 \cdot 10^6$	793.62	0.25	0.98	25
km-nonterm.5.4	$2.05 \cdot 10^6$	2,593.92	0.34	2.11	37
km-nonterm.6.5	$1.51 \cdot 10^6$	327.23	0.48	0.36	51
lamport	8	10,735.77	188.29	3,701.37	NaN
leabasicapproach	0	NaN	NaN	NaN	NaN
leaconflictset	0	NaN	NaN	NaN	NaN
lifo	$7.37 \cdot 10^6$	6,275.12	$2.55 \cdot 10^{-2}$	2.42	NaN
manufacturing	0	NaN	NaN	NaN	NaN
mesh2x2	$1.49 \cdot 10^6$	5,910.92	0.35	8.96	NaN
mesh3x2	$6.78 \cdot 10^5$	1,800.24	0.7	7.7	NaN
moesi	$4.48 \cdot 10^6$	5,557.28	$1.54 \cdot 10^{-2}$	4.55	NaN
moesi5	$1.65 \cdot 10^6$	6,573.15	$6.78 \cdot 10^{-2}$	8.55	NaN
multi-me	117	49,469.12	125.18	7,986.17	NaN
multipoll	$1.46 \cdot 10^6$	5,183.4	0.47	6.84	NaN
newdekker	0	NaN	NaN	NaN	NaN
newrtp	$5.23 \cdot 10^6$	7,038.06	0.11	5.52	NaN
peterson	0	NaN	NaN	NaN	NaN
pncsacover	0	NaN	NaN	NaN	NaN
pncsasemiliv	0	NaN	NaN	NaN	NaN
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	0	NaN	NaN	NaN	NaN
simplejavaexample	0	NaN	NaN	NaN	NaN
transthesis	0	NaN	NaN	NaN	NaN

Table C.17.: Benchmark data for P in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	$4.04 \cdot 10^6$	5,619.97	0.16	4.38	NaN
basicextransfer	$6.84 \cdot 10^6$	2,666.84	$7.66 \cdot 10^{-2}$	2.49	NaN
bingham-h250	0	NaN	NaN	NaN	NaN
consprod	0	NaN	NaN	NaN	NaN
consprod2	0	NaN	NaN	NaN	NaN
csm	0	NaN	NaN	NaN	NaN
csm-broad	64	$4.1 \cdot 10^5$	25.92	50,857.88	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$8.8 \cdot 10^5$	25.62	0.81	0.6	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	0	NaN	NaN	NaN	NaN
fms	1	0.62	0.62	0	NaN
fms2	1	96,514.52	96,514.52	0	NaN
german	0	NaN	NaN	NaN	NaN
glotter	0	NaN	NaN	NaN	NaN
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	1	14.46	14.46	0	453
km-nonterm.4.3	$2.57 \cdot 10^6$	2,029.26	0.25	1.54	25
km-nonterm.5.4	$1.98 \cdot 10^6$	294.55	0.35	0.37	37
km-nonterm.6.5	$1.51 \cdot 10^6$	317.17	0.47	0.35	51
lamport	1	8,463.1	8,463.1	0	NaN
leabasicapproach	1	12.19	12.19	0	5
leaconflictset	0	NaN	NaN	NaN	NaN
lifo	$7.41 \cdot 10^6$	5,260.63	$2.51 \cdot 10^{-2}$	3.24	NaN
manufacturing	0	NaN	NaN	NaN	NaN
mesh2x2	0	NaN	NaN	NaN	NaN
mesh3x2	0	NaN	NaN	NaN	NaN
moesi	$7.2 \cdot 10^6$	5,039.68	$1.7 \cdot 10^{-2}$	2.72	NaN
moesi5	$1.61 \cdot 10^6$	5,669.09	$6.79 \cdot 10^{-2}$	6.43	NaN
multi-me	14	$4.95 \cdot 10^5$	99.17	$1.27 \cdot 10^5$	NaN
multipoll	0	NaN	NaN	NaN	NaN
newdekker	0	NaN	NaN	NaN	NaN
newrtp	$3.53 \cdot 10^6$	4,801.41	0.18	4.18	NaN
peterson	0	NaN	NaN	NaN	NaN
pncsacover	0	NaN	NaN	NaN	NaN
pncsasemiliv	0	NaN	NaN	NaN	NaN
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	0	NaN	NaN	NaN	NaN
simplejavaexample	0	NaN	NaN	NaN	NaN
transthesis	0	NaN	NaN	NaN	NaN

Table C.18.: Benchmark data for A in Sect. 7.2.3

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T
basic-me	$4.2 \cdot 10^6$	4,670.94	0.16	3.12	NaN
basicextransfer	$6.9 \cdot 10^6$	2,941.33	$7.69 \cdot 10^{-2}$	2.49	NaN
bingham-h250	0	NaN	NaN	NaN	NaN
consprod	0	NaN	NaN	NaN	NaN
consprod2	0	NaN	NaN	NaN	NaN
csm	1	1.52	1.52	0	NaN
csm-broad	250	$2.48 \cdot 10^5$	19.25	18,755.54	NaN
delegatebuffer	0	NaN	NaN	NaN	NaN
delegatebuffer.15.1	0	NaN	NaN	NaN	NaN
delegatebuffer.16.1	0	NaN	NaN	NaN	NaN
efm	$1.01 \cdot 10^6$	38.16	0.67	0.6	NaN
examplelea	0	NaN	NaN	NaN	NaN
ext-rw	0	NaN	NaN	NaN	NaN
ext-rw-smallconsts	0	NaN	NaN	NaN	NaN
fms	6	18.63	0.67	6.72	NaN
fms2	1	2,253.25	2,253.25	0	NaN
german	0	NaN	NaN	NaN	NaN
glotter	0	NaN	NaN	NaN	NaN
hts	0	NaN	NaN	NaN	NaN
java	0	NaN	NaN	NaN	NaN
java.10.0	0	NaN	NaN	NaN	NaN
java.11.0	0	NaN	NaN	NaN	NaN
java2	0	NaN	NaN	NaN	NaN
java2.10.2	0	NaN	NaN	NaN	NaN
java2.11.2	0	NaN	NaN	NaN	NaN
kanban	1	20.67	20.67	0	415
km-nonterm.4.3	$2.63 \cdot 10^6$	4,981.67	0.25	3.25	25
km-nonterm.5.4	$2.05 \cdot 10^6$	551.97	0.34	0.47	37
km-nonterm.6.5	$1.53 \cdot 10^6$	283.67	0.48	0.31	51
lamport	0	NaN	NaN	NaN	NaN
leabasicapproach	0	NaN	NaN	NaN	NaN
leaconflictset	0	NaN	NaN	NaN	NaN
lifo	$7.17 \cdot 10^6$	6,559.47	$2.42 \cdot 10^{-2}$	3.93	NaN
manufacturing	0	NaN	NaN	NaN	NaN
mesh2x2	0	NaN	NaN	NaN	NaN
mesh3x2	0	NaN	NaN	NaN	NaN
moesi	$7.76 \cdot 10^6$	4,781.08	$1.54 \cdot 10^{-2}$	2.31	NaN
moesi5	$1.63 \cdot 10^6$	3,665.66	$6.76 \cdot 10^{-2}$	3.71	NaN
multi-me	5	$1.16 \cdot 10^5$	187.82	46,337.6	NaN
multipoll	1	301.32	301.32	0	NaN
newdekker	0	NaN	NaN	NaN	NaN
newrtp	$3.51 \cdot 10^6$	2,352.83	0.18	3.76	NaN
peterson	0	NaN	NaN	NaN	NaN
pncsacover	0	NaN	NaN	NaN	NaN
pncsasemiliv	0	NaN	NaN	NaN	NaN
queuedbusyflag	0	NaN	NaN	NaN	NaN
read-write	0	NaN	NaN	NaN	NaN
simplejavaexample	0	NaN	NaN	NaN	NaN
transthesis	0	NaN	NaN	NaN	NaN

Table C.19.: Benchmark data for $\emptyset$ in Sect. 7.2.3

C.3. Tool Comparison

In the following pages, we present most of the benchmark results that went into the comparison of the five tools (see Sect. 7.2 on p. 207 for the experimental set-up). A more detailed summary of the benchmark results can be found at
`http://csd.informatik.uni-oldenburg.de/~critter/bw-results.tar.gz`.

Framework The reference implementation of our framework.
`http://csd.informatik.uni-oldenburg.de/~critter/bw.tar.gz`
The program took 4 days and 15.68 hours to complete this benchmark set.

Petruchio/BW Our (older) coverability checker built into the PETRUCHIO tool.
`http://csd.informatik.uni-oldenburg.de/~critter/petruchio.tar.gz`
The program took 4 days and 17.27 hours to complete this benchmark set.

MIST2 `https://github.com/pierreganty/mist/`
The program took 9 days and 10.77 hours to complete this benchmark set.

BFC 2.0 `http://www.cprover.org/bfc/`
The program took 4 days and 11.72 hours to complete this benchmark set.

BFC 1.0 `http://www.cprover.org/bfc/`
The program took 5 days and 0.72 hours to complete this benchmark set.

Unfortunately, the MIST2 tool was unable to solve the coverability problem described in the HTS case study within acceptable time constraints and we stopped it after it ran for a little over 9 weeks. In total—not counting the 9 weeks just mentioned—, the benchmark set took 28 days and 7.92 hours to complete.

The tables contain the following columns (time measured in milliseconds).

Model Name of the PN or PNT model, i.e. a coverability problem.

Iter. Number of successful benchmark iterations.

Max. ms Maximal time the tool needed to solve the coverability problem.

Med. ms Median time the tool needed to solve the coverability problem.

σ **ms** Standard deviation of the time the tool needed to solve the coverability problem.

Med. T Median length of the found trace if a target state was coverable (else the table cell reads NaN). As the lengths of the found traces depend on the model's representation for each specific tool, they are not directly comparable but give a rough estimate.

E Number of unsuccessful benchmark iterations (segfault or memout over 32 GB).

If no data was collected due to a model not being supported or when nothing but errors occurred, each cell in the row for that model reads NaN.
Note that BFC 2.0 ran out of memory for four problems and BFC 1.0 had sporadic segfaults, deadlocks and memouts for many models. Also, there appears to be a bug in BFC 2.0 as it gives the wrong answer for the coverability problem of benchmark basicextransfer: While it finds a solution trace, all other tools—even BFC 1.0—agree that there is no trace connecting initial and final states. We reported the bug to the author who fixed the problem (in the module for partial-order reduction).

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T	E
basic-me	$5.78 \cdot 10^6$	12.14	$5.32 \cdot 10^{-2}$	$4.84 \cdot 10^{-2}$	NaN	0
basicextransfer	$7.22 \cdot 10^6$	25.58	$8.78 \cdot 10^{-2}$	$7.21 \cdot 10^{-2}$	NaN	0
bingham-h250	2,146	16,452.09	2,570.65	601.71	NaN	0
consprod	$1.24 \cdot 10^6$	71.7	1.83	0.34	NaN	0
consprod2	$1.79 \cdot 10^6$	11.38	0.2	$9.47 \cdot 10^{-2}$	NaN	0
csm	$2.78 \cdot 10^6$	12.01	0.13	$6.84 \cdot 10^{-2}$	NaN	0
csm-broad	$2.41 \cdot 10^6$	15.08	0.81	0.2	NaN	0
delegatebuffer	3	$1.81 \cdot 10^7$	$1.81 \cdot 10^7$	$4.28 \cdot 10^5$	NaN	0
delegatebuffer.15.1	3,383	2,566.39	2,098.48	114.22	17	0
delegatebuffer.16.1	1,361	6,198.48	5,179.96	275.01	18	0
efm	$4.01 \cdot 10^6$	89.89	0.41	0.16	NaN	0
examplelea	10,884	1,168.59	637.88	113.74	NaN	0
ext-rw	$1.56 \cdot 10^6$	12.04	0.31	0.13	NaN	0
ext-rw-smallconsts	$1.46 \cdot 10^6$	10.84	0.33	0.13	NaN	0
fms	$2.19 \cdot 10^6$	12.26	0.24	0.13	NaN	0
fms2	$2.01 \cdot 10^6$	11.57	0.25	0.13	NaN	0
german	$2.6 \cdot 10^6$	70.19	0.14	$8.5 \cdot 10^{-2}$	NaN	0
glotter	35,249	257.93	126.93	8.46	7	0
hts	64	$3.3 \cdot 10^5$	76,076.83	74,535.36	29	0
java	2,712	3,761.29	2,620.75	281.89	15	0
java.10.0	9,513	950.7	738.42	47.82	14	0
java.11.0	16,797	580.38	383.88	22.67	16	0
java2	584	17,421.63	12,230.14	1,135.71	NaN	0
java2.10.2	9,073	962.48	774.05	50.55	16	0
java2.11.2	5,616	1,593.59	1,233.42	88.69	16	0
kanban	$1.5 \cdot 10^6$	140.66	2.04	0.53	84	0
km-nonterm.4.3	$4.46 \cdot 10^6$	225.81	0.31	0.17	25	0
km-nonterm.5.4	$4.39 \cdot 10^6$	205.52	0.37	0.19	85	0
km-nonterm.6.5	$4.15 \cdot 10^6$	17.63	0.52	0.17	109	0
lamport	$3.26 \cdot 10^6$	12.96	$9.35 \cdot 10^{-2}$	$6.2 \cdot 10^{-2}$	NaN	0
leabasicapproach	$2.22 \cdot 10^6$	60.47	0.23	0.11	5	0
leaconflictset	73,308	155.81	88.61	8.59	20	0
lifo	$2.64 \cdot 10^6$	147.76	0.36	0.12	NaN	0
manufacturing	$4.26 \cdot 10^6$	153.53	0.11	$9.58 \cdot 10^{-2}$	NaN	0
mesh2x2	$1.27 \cdot 10^6$	12.45	0.62	0.34	NaN	0
mesh3x2	$6.79 \cdot 10^5$	70.18	2.11	1.09	NaN	0
moesi	$2.37 \cdot 10^6$	61.79	0.13	$8.24 \cdot 10^{-2}$	NaN	0
moesi5	$4.42 \cdot 10^5$	64.79	2.77	1.22	NaN	0
multi-me	$3.26 \cdot 10^6$	68.62	$9.94 \cdot 10^{-2}$	$7.34 \cdot 10^{-2}$	NaN	0
multipoll	$1.88 \cdot 10^6$	80.73	0.69	0.32	NaN	0
newdekker	$2.34 \cdot 10^6$	11.35	0.15	$7.52 \cdot 10^{-2}$	NaN	0
newrtp	$3.73 \cdot 10^6$	12.16	$7.21 \cdot 10^{-2}$	$4.83 \cdot 10^{-2}$	NaN	0
peterson	$2.83 \cdot 10^6$	14.53	0.18	$8.75 \cdot 10^{-2}$	NaN	0
pncsacover	$2.95 \cdot 10^5$	168.49	17.81	1.46	40	0
pncsasemiliv	$1.01 \cdot 10^6$	161.75	0.8	0.42	10	0
queuedbusyflag	747	12,781.91	9,535.59	624.05	NaN	0
read-write	$2.12 \cdot 10^6$	145.17	1.2	0.29	NaN	0
simplejavaexample	$7.19 \cdot 10^5$	84.97	3.41	0.48	11	0
transthesis	$1.15 \cdot 10^5$	583.62	34.35	7.79	NaN	0

Table C.20.: Benchmark data for Framework in Sect. 7.2.4

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T	E
basic-me	$4.87 \cdot 10^5$	1,479.14	0.12	2.36	NaN	0
basicextransfer	$1.83 \cdot 10^6$	1,220.93	0.18	1.43	NaN	0
bingham-h250	991	13,059.12	9,089.99	2,718.81	NaN	0
consprod	$1.66 \cdot 10^6$	351.72	2.3	0.79	NaN	0
consprod2	$4.79 \cdot 10^5$	3,184.43	0.21	6.62	NaN	0
csm	$4.83 \cdot 10^5$	1,448.23	0.12	5.44	NaN	0
csm-broad	$2.18 \cdot 10^6$	292.31	1.32	0.65	NaN	0
delegatebuffer	3	$1.86 \cdot 10^7$	$1.76 \cdot 10^7$	$2.17 \cdot 10^6$	NaN	0
delegatebuffer.15.1	1,882	5,478.67	3,803.8	302.57	17	0
delegatebuffer.16.1	599	17,465.35	11,950.25	1,104.46	18	0
efm	$3.26 \cdot 10^6$	986.21	0.29	1.51	NaN	0
examplelea	665	32,851.13	10,951.63	8,796.13	NaN	0
ext-rw	$4.77 \cdot 10^5$	1,862.91	0.26	4.9	NaN	0
ext-rw-smallconsts	$4.77 \cdot 10^5$	1,217.59	0.25	3.86	NaN	0
fms	$9.37 \cdot 10^5$	2,142.56	0.2	3.87	NaN	0
fms2	$9.31 \cdot 10^5$	3,368.74	0.22	5.78	NaN	0
german	$1.67 \cdot 10^6$	514.83	0.17	1.71	NaN	0
glotter	$1.76 \cdot 10^5$	447.51	35.89	13.9	7	0
hts	89	$1.5 \cdot 10^5$	81,275.65	13,507.94	29	0
java	15,696	906.58	455.38	45.99	15	0
java.10.0	12,111	999.45	591.08	50.81	14	0
java.11.0	31,940	484.3	221.94	20.34	15	0
java2	1,480	7,896.15	4,789.06	583.89	NaN	0
java2.10.2	11,468	1,069.87	624.57	51.58	14	0
java2.11.2	9,741	1,251.11	733.6	65.87	15	0
kanban	$1 \cdot 10^7$	5,594.82	0.52	6.13	109	0
km-nonterm.4.3	$3.28 \cdot 10^6$	677.12	0.25	0.94	18	0
km-nonterm.5.4	$3.17 \cdot 10^6$	1,797.2	0.32	1.36	28	0
km-nonterm.6.5	$3.03 \cdot 10^6$	506.38	0.42	0.7	40	0
lamport	$4.84 \cdot 10^5$	1,280.61	0.13	2.29	NaN	0
leabasicapproach	$4.81 \cdot 10^5$	1,676.6	0.23	5.46	5	0
leaconflictset	$2.69 \cdot 10^5$	193.96	25.91	2.19	20	0
lifo	$1.8 \cdot 10^6$	2,224.24	0.17	2.92	NaN	0
manufacturing	$4.86 \cdot 10^5$	2,294.24	0.12	4.71	NaN	0
mesh2x2	$9.03 \cdot 10^5$	766.86	0.4	3.09	NaN	0
mesh3x2	$1.29 \cdot 10^6$	2,437.92	1.2	5.73	NaN	0
moesi	$9.39 \cdot 10^5$	1,553.36	0.16	4.14	NaN	0
moesi5	$1.09 \cdot 10^6$	5,070.46	2.18	9.59	NaN	0
multi-me	$2.44 \cdot 10^5$	3,206.84	0.14	6.68	NaN	0
multipoll	$9.39 \cdot 10^5$	1,601.95	0.19	3.43	NaN	0
newdekker	$1.35 \cdot 10^6$	217.95	2.89	2.75	NaN	0
newrtp	$4.86 \cdot 10^5$	826.7	0.1	2.82	NaN	0
peterson	$4.82 \cdot 10^5$	1,223.76	0.18	3.12	NaN	0
pncsacover	$1.13 \cdot 10^6$	406.55	4.16	1.44	35	0
pncsasemiliv	$8.82 \cdot 10^5$	1,415.6	0.56	5.36	10	0
queuedbusyflag	884	11,589.93	8,076.56	699.74	NaN	0
read-write	$8.94 \cdot 10^5$	1,039.07	0.9	2.27	NaN	0
simplejavaexample	$1.16 \cdot 10^6$	359.37	4.85	1.13	13	0
transthesis	$1.17 \cdot 10^5$	873.58	59.14	10.32	NaN	0

Table C.21.: Benchmark data for PETRUCHIO/BW in Sect. 7.2.4

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T	E
basic-me	$9.83 \cdot 10^6$	7.42	0.61	0.31	NaN	0
basicextransfer	$1.05 \cdot 10^8$	6.4	$5.3 \cdot 10^{-2}$	$9.23 \cdot 10^{-2}$	NaN	0
bingham-h250	436	16,794.03	16,529.99	156.84	NaN	0
consprod	$3.84 \cdot 10^5$	26.75	18.63	1.51	NaN	0
consprod2	$1.03 \cdot 10^7$	9.11	0.66	0.26	NaN	0
csm	$2.47 \cdot 10^5$	38.79	29.05	1.58	NaN	0
csm-broad	$6.15 \cdot 10^5$	19.63	11.44	1.19	NaN	0
delegatebuffer	3	$8.98 \cdot 10^7$	$8.65 \cdot 10^7$	$3.02 \cdot 10^6$	NaN	0
delegatebuffer.15.1	19	$3.86 \cdot 10^5$	$3.81 \cdot 10^5$	8,235.38	13	0
delegatebuffer.16.1	7	$1.2 \cdot 10^6$	$1.2 \cdot 10^6$	13,107.87	13	0
efm	$7.22 \cdot 10^6$	10.93	0.97	0.4	NaN	0
examplelea	834	9,210.04	8,647.29	249.09	NaN	0
ext-rw	5	$1.74 \cdot 10^6$	$1.74 \cdot 10^6$	73,060.2	NaN	0
ext-rw-smallconsts	98,164	94.57	72.14	4.97	NaN	0
fms	$5.71 \cdot 10^8$	2.57	$9 \cdot 10^{-3}$	$3.35 \cdot 10^{-2}$	NaN	0
fms2	158	47,913.54	45,917.27	1,560.36	NaN	0
german	$1.42 \cdot 10^7$	8.58	0.48	0.23	NaN	0
glotter	3	$7.6 \cdot 10^7$	$7.36 \cdot 10^7$	$2.09 \cdot 10^6$	7	0
hts	NaN	NaN	NaN	NaN	NaN	NaN
java	2,692	2,850.81	2,680.04	73.65	14	0
java.10.0	112	70,656.45	65,229.55	3,474.03	10	0
java.11.0	213	35,641.2	34,611.14	1,490.5	10	0
java2	$6.28 \cdot 10^5$	22.96	11.32	1.69	NaN	0
java2.10.2	100	75,962.04	72,885.62	2,441.72	10	0
java2.11.2	93	79,674.6	77,197.61	1,544	10	0
kanban	3	$4.27 \cdot 10^6$	$4.09 \cdot 10^6$	$1.6 \cdot 10^5$	48	0
km-nonterm.4.3	$1.74 \cdot 10^6$	15.51	3.81	1	12	0
km-nonterm.5.4	$1.29 \cdot 10^6$	14.82	5.27	0.88	20	0
km-nonterm.6.5	$9.22 \cdot 10^5$	15.48	7.49	1.01	30	0
lamport	$7.07 \cdot 10^6$	8.98	0.97	0.25	NaN	0
leabasicapproach	$6.4 \cdot 10^5$	17.46	11	1.02	4	0
leaconflictset	8,871	938.54	809.63	36.94	15	0
lifo	NaN	NaN	NaN	NaN	NaN	NaN
manufacturing	560	13,245.45	12,956.92	342.12	NaN	0
mesh2x2	9,363	903.03	765.86	26.47	NaN	0
mesh3x2	504	14,714.71	14,335.89	237.25	NaN	0
moesi	NaN	NaN	NaN	NaN	NaN	NaN
moesi5	NaN	NaN	NaN	NaN	NaN	NaN
multi-me	$3.91 \cdot 10^5$	26.04	18.41	1.35	NaN	0
multipoll	3,027	2,534.6	2,381.39	61.24	NaN	0
newdekker	$6.25 \cdot 10^5$	21.8	11.34	1.24	NaN	0
newrtp	$4.3 \cdot 10^8$	2.24	$1.4 \cdot 10^{-2}$	$2.92 \cdot 10^{-2}$	NaN	0
peterson	$6.41 \cdot 10^5$	20.91	10.92	1.37	NaN	0
pncsacover	152	49,031.9	47,527.11	794.75	32	0
pncsasemiliv	13,532	595.39	534.47	22.77	10	0
queuedbusyflag	3	$5.89 \cdot 10^6$	$5.56 \cdot 10^6$	$3.01 \cdot 10^5$	NaN	0
read-write	52,977	160.62	135.71	5.5	NaN	0
simplejavaexample	23,965	371.11	302.46	16.92	10	0
transthesis	82,333	130.44	86.43	7.28	NaN	0

Table C.22.: Benchmark data for MIST2 in Sect. 7.2.4

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T	E
basic-me	$3.05 \cdot 10^6$	145.94	2.34	0.23	NaN	0
basicextransfer	$3.32 \cdot 10^6$	338.23	2.15	0.31	3	0
bingham-h250	79	94,802.06	91,978.45	1,003.15	NaN	0
consprod	$8.98 \cdot 10^5$	109.72	8	0.28	NaN	0
consprod2	$9.31 \cdot 10^5$	51.57	7.71	0.24	NaN	0
csm	$2.07 \cdot 10^6$	216.89	3.46	0.31	NaN	0
csm-broad	$2.42 \cdot 10^6$	333.19	2.96	0.4	NaN	0
delegatebuffer	0	NaN	NaN	NaN	NaN	3
delegatebuffer.15.1	310	35,188.18	22,495.12	2,831.31	40	0
delegatebuffer.16.1	37	$2.64 \cdot 10^5$	$1.9 \cdot 10^5$	22,819.99	39	0
efm	$3.09 \cdot 10^6$	135.48	2.31	0.23	NaN	0
examplelea	3	$3.87 \cdot 10^6$	$3.86 \cdot 10^6$	22,784.61	NaN	0
ext-rw	0	NaN	NaN	NaN	NaN	3
ext-rw-smallconsts	0	NaN	NaN	NaN	NaN	3
fms	83,602	523.18	85.47	7.21	NaN	0
fms2	420	20,380.59	17,003.27	846	NaN	0
german	$2.51 \cdot 10^6$	476.98	2.84	1.32	NaN	0
glotter	$1.57 \cdot 10^5$	543.2	45.54	4.72	15	0
hts	2,059	12,398.89	3,172.68	1,098.57	31	0
java	$2.62 \cdot 10^5$	107.31	25.31	10.5	30	0
java.10.0	2,381	4,344.01	2,995.92	116.63	34	0
java.11.0	285	30,681.37	25,100.68	1,030.3	35	0
java2	17	$4.94 \cdot 10^5$	$4.4 \cdot 10^5$	19,278.98	NaN	0
java2.10.2	2,419	3,963.38	2,934.29	144.88	30	0
java2.11.2	295	31,379.67	24,046.2	1,360.08	30	0
kanban	$1.34 \cdot 10^5$	138.53	55.45	3.98	13	0
km-nonterm.4.3	$1.47 \cdot 10^6$	178.46	4.87	0.36	20	0
km-nonterm.5.4	$1.26 \cdot 10^6$	327.29	5.67	1	30	0
km-nonterm.6.5	$1.1 \cdot 10^6$	338.8	6.46	0.97	42	0
lamport	$2.65 \cdot 10^6$	385.51	2.69	0.57	NaN	0
leabasicapproach	$2.32 \cdot 10^6$	136.82	3.09	0.28	6	0
leaconflictset	$7.52 \cdot 10^5$	53.1	9.64	1.37	33	0
lifo	NaN	NaN	NaN	NaN	NaN	NaN
manufacturing	$2.76 \cdot 10^6$	218.48	2.58	0.29	NaN	0
mesh2x2	104	$1.41 \cdot 10^5$	85,046.49	32,483.66	NaN	0
mesh3x2	3	$8.12 \cdot 10^6$	$7.73 \cdot 10^6$	$3.82 \cdot 10^5$	NaN	0
moesi	NaN	NaN	NaN	NaN	NaN	NaN
moesi5	NaN	NaN	NaN	NaN	NaN	NaN
multi-me	$2.21 \cdot 10^6$	222.16	3.24	0.38	NaN	0
multipoll	4,398	1,836.2	1,635.16	35.69	NaN	0
newdekker	$1.99 \cdot 10^6$	213.01	3.58	0.45	NaN	0
newrtp	$3.41 \cdot 10^6$	146.63	2.09	0.26	NaN	0
peterson	$2.35 \cdot 10^6$	195.91	3.04	0.34	NaN	0
pncsacover	3,873	4,312.86	1,783.81	486.75	38	0
pncsasemiliv	$1.53 \cdot 10^6$	129.39	4.68	0.26	12	0
queuedbusyflag	0	NaN	NaN	NaN	NaN	3
read-write	$2.02 \cdot 10^6$	288.16	3.55	0.36	NaN	0
simplejavaexample	$1.4 \cdot 10^6$	509.34	5.03	1.02	12	0
transthesis	41	$2.1 \cdot 10^5$	$1.77 \cdot 10^5$	8,357.4	NaN	0

Table C.23.: Benchmark data for BFC 2.0 in Sect. 7.2.4

Model	Iter.	Max. ms	Med. ms	σ ms	Med. T	E
basic-me	$2.99 \cdot 10^6$	764.49	2.37	1.1	NaN	1
basicextransfer	$2.18 \cdot 10^6$	466.04	3.17	1.23	NaN	0
bingham-h250	15	$5.35 \cdot 10^5$	$4.94 \cdot 10^5$	14,144.29	NaN	0
consprod	$1.4 \cdot 10^5$	433.67	49.61	5.71	NaN	285
consprod2	$1.78 \cdot 10^5$	560.22	38.71	5.38	NaN	335
csm	$1.08 \cdot 10^6$	4,211.59	6.42	4.22	NaN	1
csm-broad	$4.66 \cdot 10^5$	524.95	14.39	2.57	NaN	0
delegatebuffer	0	NaN	NaN	NaN	NaN	3
delegatebuffer.15.1	157	52,650.14	45,633.79	2,200.56	40	0
delegatebuffer.16.1	21	$5.07 \cdot 10^5$	$3.38 \cdot 10^5$	44,978.54	32	0
efm	$3.17 \cdot 10^6$	507.99	2.25	0.5	NaN	1
examplelea	2,247	3,619.31	3,209.76	84.86	NaN	8
ext-rw	0	NaN	NaN	NaN	NaN	3
ext-rw-smallconsts	0	NaN	NaN	NaN	NaN	5
fms	$4.63 \cdot 10^5$	3,585.2	14.64	11.29	NaN	119
fms2	163	48,767.7	44,594.4	2,136.18	NaN	1
german	$2.53 \cdot 10^6$	991.06	2.81	0.88	NaN	1
glotter	76,044	234.58	73.32	30.94	13	46
hts	286	$1.36 \cdot 10^5$	21,165.79	19,761.4	31	0
java	14,915	100.34	47.73	9.48	22	183
java.10.0	347	26,312.76	20,787.34	2,017.85	36	1
java.11.0	50	$1.7 \cdot 10^5$	$1.45 \cdot 10^5$	11,604.87	35	1
java2	4,184	5,244.54	1,575.98	584.29	NaN	46
java2.10.2	346	29,016.35	20,999.18	2,064.9	32	2
java2.11.2	54	$1.54 \cdot 10^5$	$1.35 \cdot 10^5$	9,127.46	33	0
kanban	5,988	8,935.39	719.58	1,286.96	18	284
km-nonterm.4.3	$1.27 \cdot 10^6$	128.34	5.75	0.58	20	0
km-nonterm.5.4	$6.17 \cdot 10^5$	544.73	8.3	1.89	30	9,379
km-nonterm.6.5	$6.73 \cdot 10^5$	397.9	10.76	2.37	42	0
lamport	$2.51 \cdot 10^6$	196.22	2.84	0.29	NaN	1
leabasicapproach	$1.6 \cdot 10^6$	208.92	3.1	1.86	6	561
leaconflictset	$2.76 \cdot 10^5$	52.25	16.19	2.16	21	803
lifo	NaN	NaN	NaN	NaN	NaN	NaN
manufacturing	$2.29 \cdot 10^6$	247.78	3.12	0.39	NaN	0
mesh2x2	8,530	6,168.31	765	640.22	NaN	298
mesh3x2	2	73,120.58	65,975.85	7,144.73	NaN	2
moesi	NaN	NaN	NaN	NaN	NaN	NaN
moesi5	NaN	NaN	NaN	NaN	NaN	NaN
multi-me	$1.38 \cdot 10^6$	293.95	5.33	0.73	NaN	107
multipoll	482	17,066.78	14,831.42	695.27	NaN	13
newdekker	$1.79 \cdot 10^6$	409.24	3.98	1	NaN	1
newrtp	$3.63 \cdot 10^6$	677.56	1.96	0.78	NaN	0
peterson	$2.23 \cdot 10^6$	553.92	3.19	0.66	NaN	166
pncsacover	2,409	8,124.96	2,843.38	903.88	38	0
pncsasemiliv	$1.46 \cdot 10^6$	131.44	4.86	0.49	12	1
queuedbusyflag	172	45,465.84	41,885.04	1,131.48	NaN	0
read-write	$4.58 \cdot 10^5$	37.44	7.89	1.24	NaN	1
simplejavaexample	$9.61 \cdot 10^5$	42.99	5.31	5.37	12	31
transthesis	129	63,475.64	55,912.98	2,062.1	NaN	0

Table C.24.: Benchmark data for BFC 1.0 in Sect. 7.2.4

Bibliography

[ABH$^+$97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 340–351. Springer Berlin Heidelberg, 1997. Cited on page 147.

[ABJ98] Parosh Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In Alan J. Hu and Moshe Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318. Springer Berlin Heidelberg, 1998. Cited on page 147.

[ABO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Texts in Computer Science. Springer Berlin Heidelberg, 3rd edition, 2009. 2nd Printing. Cited on pages 43, 45, 46, 47, 48, 76, and 88.

[AČJT96] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-

state systems. In Edmund Melson Clarke, editor, *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, LICS '96, pages 313–323, Washington, DC, USA, 1996. IEEE Computer Society. Cited on pages 4, 5, 6, 30, 51, and 52.

[AIN04] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. Sat-solving the coverability problem for petri nets. *Formal Methods in System Design*, 24:25–43, 2004. Cited on page 6.

[AJ93] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *Proceedings of the 8th IEEE International Symposium on Logic in Computer Science (LICS'93)*, pages 160–170. IEE, June 1993. Cited on page 40.

[AJ94] Parosh Aziz Abdulla and Bengt Jonsson. Undecidable verification problems for programs with unreliable channels. In Serge Abiteboul and Eli Shamir, editors, *Proceedings of the 21st International Colloquium on Automata, Languages and Programming (ICALP'94)*, volume 820 of *Lecture Notes in Computer Science*, pages 316–327. Springer Berlin Heidelberg, 1994. Cited on page 40.

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, June 1996. Cited on pages 40 and 41.

[AJKP98] Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification. In Alan J. Hu and Moshe Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 379–390. Springer Berlin Heidelberg, 1998. Cited on pages 52, 81, 147, 148, 151, and 208.

[AKP97] Parosh Aziz Abdulla, Mats Kindahl, and Doron Peled. An improved search strategy for lossy channel systems. In

Tadanori Mizuno, Nori Shiratori, Teruo Hegashino, and Atsushi Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE X/PSTV XVII'97)*, IFIP Advances in Information and Communication Technology, pages 251–264. Springer US, 1997. Cited on pages 40, 151, and 208.

[Ame84] Khaled A. Amer. Equationally complete classes of commutative monoids with monus. *Algebra Universalis*, 18(1):129–131, 1984. Cited on page 141.

[BBS06] Christel Baier, Nathalie Bertrand, and Philippe Schnoebelen. On computing fixpoints in well-structured regular model checking, with applications to lossy channel systems. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'06)*, volume 4246 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin Heidelberg, 2006. Cited on page 228.

[BF99] Béatrice Bérard and Laurent Fribourg. Reachability analysis of (timed) petri nets using real arithmetic. In Jos Baeten and Sjouke Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 178–193. Springer Berlin Heidelberg, 1999. Cited on page 246.

[BG11] Laura Bozzelli and Pierre Ganty. Complexity analysis of the backward coverability algorithm for VASS. In Giorgio Delzanno and Igor Potapov, editors, *Proceedings of the 5th International Workshop on Reachability Problems (RP'11)*, volume 6945 of *Lecture Notes in Computer Science*, pages 96–109. Springer Berlin Heidelberg, 2011. Cited on pages 6 and 30.

[BH05] Jesse Bingham and Alan J. Hu. Empirically efficient verification for a class of infinite-state systems. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume

3440 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin Heidelberg, 2005. Cited on pages 6, 161, 220, and 230.

[Bin05] Jesse Bingham. A new approach to upward-closed set backward reachability analysis. *Electronic Notes in Theoretical Computer Science*, 138:37–48, December 2005. Cited on pages 6, 161, and 220.

[Bir40] Garrett Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, 1st edition, 1940. Cited on page 17.

[BJLZ12] Frank S. de Boer, Mahdi M. Jaghoori, Cosimo Laneve, and Gianluigi Zavattaro. Decidability problems for actor systems. In Maciej Koutny and Irek Ulidowski, editors, *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12)*, volume 7454 of *Lecture Notes in Computer Science*, pages 562–577. Springer Berlin Heidelberg, 2012. Cited on page 229.

[Blo08] Joshua Bloch. *Effective Java: A Programming Language Guide*. The Java Series ... from the Source. Addison-Wesley Longman, Amsterdam, 2nd edition, May 2008. Cited on pages 195 and 256.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*, PLDI '01, pages 203–213. ACM Press, 2001. Cited on page 256.

[BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer Berlin Heidelberg, 2002. 10.1007/3-540-46002-0_12. Cited on page 256.

[BR99] Arnulf Braatz and Arno Ritter. Referenzfallstudie Produktionstechnik v1.3: Spezifikation des verteilten Steuerungskonzeptes für den holonischen Materialfluß in einem werkstattorientierten Fertigungssystem auf der Basis autonomer, freifahrender Transportsysteme (Holonischer Materialfluß). Technical report, IFF Universität Stuttgart & Fraunhofer IPA, Stuttgart, March 1999. Cited on pages 206 and 239.

[BR01] Arnulf Braatz and Arno Ritter. Referenzfallstudie Produktionstechnik (PA) v2.0: Spezifikation des verteilten Steuerungskonzeptes für den holonischen Materialfluß in einem werkstattorientierten Fertigungssystem auf der Basis autonomer, freifahrender Transportsysteme (Holonischer Materialfluß). Technical report, IFF Universität Stuttgart & Fraunhofer IPA, Stuttgart, September 2001. Cited on pages 206 and 239.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986. Cited on page 160.

[BSW69] Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969. Cited on page 41.

[CFS11] Pierre Chambart, Alain Finkel, and Sylvain Schmitz. Forward analysis and model checking for trace bounded WSTS. In Lars M. Kristensen and Laure Petrucci, editors, *Proceedings of the 32nd International Conference on the Applications and Theory of Petri Nets (ATPN'11)*, volume 6709 of *Lecture Notes in Computer Science*, pages 49–68. Springer Berlin Heidelberg, 2011. Cited on page 230.

[CGJ+00] Edmund Melson Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In Allen E. Emerson and Aravinda Prasad Sistla, editors, *Proceedings of the 12th International Conference on*

Computer Aided Verification (CAV'00), volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000. Cited on page 7.

[Chi90] Giovanni Chiola. Compiling techniques for the analysis of stochastic petri nets. In Ramon Puigjaner and Dominique Potier, editors, *Proceedings of the 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation (TOOLS'90)*, pages 11–24. Springer New York, 1990. Cited on page 160.

[Cia94] Gianfranco Ciardo. Petri nets with marking-dependent arc cardinality: Properties and analysis. In *Proceedings of the 15th International Conference on the Application and Theory of Petri Nets (ATPN'94)*, pages 179–198, London, UK, 1994. Springer Berlin Heidelberg. Cited on pages 38 and 147.

[CM97] Gianfranco Ciardo and Andrew S. Miner. Storage alternatives for large structured state spaces. In Raymond A. Marie, Brigitte Plateau, Maria Calzarossa, and Gerardo Rubino, editors, *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'97)*, volume 1245 of *Lecture Notes in Computer Science*, pages 44–57. Springer Berlin Heidelberg, 1997. Cited on pages xvii, 160, 251, 252, and 253.

[Cor96] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996. Cited on page 220.

[CS91] José Colom and Manuel Silva. Convex geometry and semiflows in p/t nets. a comparative study of algorithms for computation of minimal p-semiflows. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 – Selected Papers of the Proceedings of the 10th International Conference on the Applications and Theory of Petri Nets (ATPN'89)*, volume 483 of *Lecture Notes in Computer Science*, pages 79–112. Springer Berlin Heidelberg, 1991. Cited on page 147.

[CT96] Gianfranco Ciardo and Marco Tilgner. On the use of kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96–35, Institute for Computer Applications in Science and Engineering, Hampton, VA, USA, 1996. Cited on page 250.

[DFP06] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. Directed model checking with distance-preserving abstractions. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 2006. Cited on page 156.

[DFS98] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer Berlin Heidelberg, 1998. Cited on page 38.

[Dic13] Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4):413–422, 1913. Cited on pages 17 and 21.

[DP08] Rayna Dimitrova and Andreas Podelski. Is lazy abstraction a decision procedure for broadcast protocols? In Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck, editors, *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*, volume 4905 of *Lecture Notes in Computer Science*, pages 98–111. Springer Berlin Heidelberg, 2008. Cited on page 7.

[DR00] Giorgio Delzanno and Jean-François Raskin. Symbolic representation of upward-closed sets. In Susanne Graf and Michael Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume

1785 of *Lecture Notes in Computer Science*, pages 426–441. Springer Berlin Heidelberg, 2000. Cited on page 160.

[DRV01] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Attacking symbolic state explosion. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 298–310. Springer Berlin Heidelberg, 2001. Cited on pages 81, 143, 215, and 220.

[DRV02] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Towards the automated verification of multithreaded java programs. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 105–117. Springer Berlin Heidelberg, 2002. Cited on pages 37, 160, and 220.

[DT88] Mario D'Anna and Sebastiano Trigila. Concurrent system analysis using petri nets: An optimized algorithm for finding net invariants. *Computer Communications*, 11(4):215–220, 1988. Cited on page 147.

[ELLL01] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In Matthew Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking Software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer Berlin Heidelberg, 2001. Cited on page 158.

[ELLL04] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2–3):247–267, 2004. Cited on page 158.

[Fin87] Alain Finkel. A generalization of the procedure of karp and miller to well structured transition systems. In

Thomas Ottmann, editor, *Proceedings of the 14th International Colloquium on Automata, Languages and Programming (ICALP'87)*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer Berlin Heidelberg, 1987. Cited on page 4.

[Fin93] Alain Finkel. The minimal coverability graph for petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993 – Selected Papers of the Proceedings of the 12th International Conference on the Application and Theory of Petri Nets (ATPN'93)*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer Berlin Heidelberg, 1993. Cited on pages xvii, 219, 246, and 247.

[FRSV03] Alain Finkel, Jean-François Raskin, Mathias Samuelides, and Laurent Van Begin. Monotonic extensions of petri nets: Forward and backward search revisited. *Electronic Notes in Theoretical Computer Science*, 68(6):85–106, 2003. Infinity 2002, 4th International Workshop on Verification of Infinite-State Systems (CONCUR 2002 Satellite Workshop). Cited on pages 6 and 160.

[FS01] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, April 2001. Cited on pages 4, 20, 51, 52, and 230.

[Gan07] Pierre Ganty. *The Fixpoint Checking Problem: An Abstraction Refinement Perspective.* PhD thesis, Université Libre de Bruxelles, Faculté des Sciences, Département d'Informatique, September 2007. Cited on page 6.

[GBB+06] Brian Goetz, Joshua Bloch, Joseph Browbeer, Doug Lea, David Holmes, and Tim Peierly. *Java Concurrency in Practice.* Addison-Wesley Longman, Amsterdam, May 2006. Cited on page 256.

[Gee07] Gilles Geeraerts. *Coverability and Expressiveness Properties of Well-structured Transition Systems.* PhD thesis, Université Libre de Bruxelles, Faculté des Sciences, Département d'Informatique, May 2007. Cited on page 246.

[GHJ94] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reausable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st edition, October 1994. Cited on page 255.

[GIE88] GIE CB. Protocole normalisé de connexion au système d'autorisation: Spécifications PNCSA version 2. Document C, Communcation OSI, 1988. Cited on page 246.

[GMV$^+$07] Pierre Ganty, Cédric Meuter, Laurent Van Begin, Gabriel Kalyon, Jean-François Raskin, and Giorgio Delzanno. Symbolic data structure for sets of k-uples of integers. Technical Report 570, Université Libre de Bruxelles, 2007. Cited on pages 160 and 215.

[Gra97] Bernd Grahlmann. The PEP tool. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443. Springer Berlin Heidelberg, 1997. Cited on page 204.

[Gra98] Bernd Grahlmann. The state of PEP. In Armando Martin Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 522–526. Springer Berlin Heidelberg, 1998. Cited on page 204.

[Gri07] Philipp Gringel. Modellierung und Verifikation eines holonischen Transportsystems mit dem π-Kalkül. Bachelor's thesis, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, November 2007. Cited on page 240.

[GRV05] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, enlarge and check... made efficient. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 394–407. Springer Berlin Heidelberg, 2005. Cited on pages 6, 215, 220, and 246.

[GRV06a] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. A complete abstract interpretation framework for coverability properties of wsts. In Ernest Allen Emerson and Kedar Namjoshi, editors, *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2006. Cited on pages 7 and 215.

[GRV06b] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, enlarge and check: New algorithms for the coverability problem of WSTS. *Journal of Computer and System Sciences*, 72(1):180–203, 2006. Cited on pages 6, 147, and 220.

[GRV07] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for petri nets. In Kedar Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin Heidelberg, 2007. Cited on page 6.

[GRV08] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few: Automatic abstraction refinement for petri nets. *Fundamenta Informaticae*, 88(3):275–305, December 2008. Cited on pages 6, 193, and 215.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin Heidelberg, 1997. Cited on page 256.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of

Lecture Notes in Computer Science, pages 460–463. Springer Berlin Heidelberg, 1997. Cited on page 220.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952. Cited on pages 17, 18, 22, 27, and 41.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, volume 37(1) of *ACM SIGPLAN Notices*, pages 58–70. ACM Press, 2002. Cited on page 7.

[HKQ03] Thomas A. Henzinger, Orna Kupferman, and Shaz Qadeer. From *Pre*historic to *Post*modern symbolic model checking. *Formal Methods in System Design*, 23(3):303–327, 2003. Cited on page 143.

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968. Cited on page 156.

[KHDB06] Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an ai planning heuristic for directed model checking. In Antti Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer Berlin Heidelberg, 2006. Cited on page 156.

[KHL08] Sebastian Kupferschmid, Jörg Hoffmann, and Kim Guldstrand Larsen. Fast directed model checking via russian doll abstraction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin Heidelberg, 2008. Cited on page 156.

[KKW10] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 645–659. Springer Berlin Heidelberg, 2010. Cited on page 37.

[KKW12] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Efficient coverability analysis by proof minimization. In Maciej Koutny and Irek Ulidowski, editors, *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR'12)*, volume 7454 of *Lecture Notes in Computer Science*, pages 500–515. Springer Berlin Heidelberg, 2012. Cited on pages 7, 193, and 215.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. Cited on page 5.

[KMNP13] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. *Computing Research Repository (CoRR)*, abs/1301.7321, 2013. Non-reviewed version, original version submitted to CAV 2013; this is a revised version, containing more experimental results and some corrections (Feb. 22. 2013). Cited on pages 7, 220, and 230.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998. Cited on pages 167, 169, and 189.

[LCL87] Fuchun Joseph Lin, P. M. Chu, and Min Tsan Liu. Protocol verification using reachability analysis: The state space explosion problem and relief strategies. *ACM SIGCOMM Computer Communication Review*, 17(5):126–135, August 1987. Cited on page 155.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Java Series ... from the Source. Addison-

Wesley Longman, Amsterdam, 2nd edition, October 1999. Cited on pages xxiii, 195, 255, and 256.

[Lip76] Richard J. Lipton. The reachability problem requires exponential space. Technical Report 63, Department of Computer Science, Yale University, New Haven, CT, USA, 1976. Cited on page 36.

[LL00] Michael Leuschel and Helko Lehmann. Solving coverability problems of petri nets by partial deduction. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, PPDP '00, pages 268–279, New York, NY, USA, 2000. ACM. Cited on page 6.

[MBC+95] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley & Sons, November 1995. Cited on pages xvii, 250, and 251.

[MC99] Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In Susanna Donatelli and Jetty Kleijn, editors, *Proceedings of the 20th International Conference on the Application and Theory of Petri Nets (ATPN'99)*, volume 1639 of *Lecture Notes in Computer Science*, pages 691–691. Springer Berlin Heidelberg, 1999. Cited on page 160.

[Mey08] Roland Meyer. *Structural Stationarity in the π-Calculus*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, November 2008. Cited on pages 193 and 240.

[Mey09] Roland Meyer. A theory of structural stationarity in the π-calculus. *Acta Informatica*, 46:87–137, 2009. Cited on pages 8, 193, and 215.

[Mil04] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1st edition, 2004. 5th Printing. Cited on page 19.

[MJ84] Francis Lockwood Morris and Clifford B. Jones. An early program proof by alan turing. *IEEE Annals of the History of Computing*, 6(2):139–143, 1984. Cited on page 4.

[MKS09] Roland Meyer, Victor Khomenko, and Tim Strazny. A practical approach to verification of mobile systems using net unfoldings. *Fundamenta Informaticae*, 94(3–4):439–471, 2009. Special Issue on Petri Nets 2008, invited version of the ATPN 2008 paper. Cited on pages xxiii, 8, 219, 240, and 241.

[MMW13] Rupak Majumdar, Roland Meyer, , and Zilong Wang. Static provenance verification for message passing programs. In Francesco Logozzo and Manuel Fähndrich, editors, *Proceedings of the 20th International Symposium on Static Analysis (SAS'13)*, volume 7935 of *Lecture Notes in Computer Science*, pages 366–387. Springer Berlin Heidelberg, 2013. Cited on page 220.

[MORW04] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Linking CSP-OZ with UML and Java: A case study. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM'04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 267–286. Springer Berlin Heidelberg, 2004. Cited on page 240.

[MS10] Roland Meyer and Tim Strazny. Petruchio: From dynamic networks to nets. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 175–179. Springer Berlin Heidelberg, 2010. Cited on pages 8, 193, 215, and 240.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976. Cited on pages 46 and 88.

[OWW10] Olivia Oanea, Harro Wimmel, and Karsten Wolf. New algorithms for deciding the siphon-trap property. In Johan Lilius

and Wojciech Penczek, editors, *Proceedings of the 31st International Conference on the Applications and Theory of Petri Nets (ATPN'10)*, volume 6128 of *Lecture Notes in Computer Science*, pages 267–286. Springer Berlin Heidelberg, 2010. Cited on page 147.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer Berlin Heidelberg, 1993. Cited on page 147.

[Pel96] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996. Cited on page 147.

[Pel98] Doron Peled. Ten years of partial order eeduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer Berlin Heidelberg, 1998. Cited on page 148.

[Pel06] Radek Pelánek. Web portal for benchmarking explicit model checkers. Technical Report FIMU-RS-2006-03, FI MU, Faculty of Informatics, Masaryk University Brno, Czech Republic, March 2006. 39 Pages. Cited on page 220.

[Pel07] Radek Pelánek. Beem: Benchmarks for explicit model checkers. In Dragan Bošnacki and Stefan Edelkamp, editors, *Proceedings of the 14th International SPIN Workshop on Model Checking Software (SPIN'07)*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer Berlin Heidelberg, 2007. Cited on page 220.

[PP05] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*. In a Nutshell. O'Reilly Media, 2005. Cited on page 199.

[PW08] Lutz Priese and Harro Wimmel. *Petri-Netze*. eXamen.press. Springer Berlin Heidelberg, 2nd edition, 2008. Cited on page 12.

[Rac78] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978. Cited on pages 6 and 36.

[RY85] Louis Rosier and Hsu-Chun Yen. A multiparameter analysis of the boundedness problem for vector addition systems. In Lothar Budach, editor, *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT'85)*, volume 199 of *Lecture Notes in Computer Science*, pages 361–370. Springer Berlin Heidelberg, 1985. Cited on page 6.

[Sch02] Philippe Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5):251–261, 2002. Cited on pages 41 and 208.

[Sch10] Philippe Schnoebelen. Revisiting ackermann-hardness for lossy counter machines and reset petri nets. In Petr Hlinený and Antonín Kucera, editors, *Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS'10)*, volume 6281 of *Lecture Notes in Computer Science*, pages 616–628. Springer Berlin Heidelberg, 2010. Cited on pages 38 and 41.

[Sed02] Robert Sedgewick. *Algorithms in Java, Parts 1–4*. International. Addison Wesley, 3rd edition, 2002. Cited on pages 167, 169, and 189.

[SKMB90] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of the 9th International Conference on Computer-Aided Design (ICCAD'90)*, pages 92–95. IEEE Computer Society, 1990. Cited on page 160.

[SM12] Tim Strazny and Roland Meyer. An algorithmic framework for coverability in well-structured systems. In Jens Brandt and Keijo Heljanko, editors, *Proceedings of the 12th International Conference on the Application of Concurrency to*

System Design (ACSD'12), pages 173–182. IEEE Computer Society Conference Publishing Services, June 2012. Best Paper Award of ACSD. Cited on pages 8, 37, 40, 208, 215, and 228.

[STC98] Manuel Silva, Enrique Teruel, and José Colom. Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 309–373. Springer Berlin Heidelberg, 1998. Cited on pages 145 and 147.

[Str07] Tim Strazny. Entwurf und Implementierung von Algorithmen zur Berechnung von Petrinetz-Semantiken für Pi-Kalkül-Prozesse. Master's thesis, Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, July 2007. Cited on pages 8 and 193.

[Str11] Tim Strazny. Accelerating backward reachability analysis. In Paul Pettersson and Cristina Seceleanu, editors, *Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT'11)*, pages 2–4. Mälardalen University Sweden, October 2011. Technical report 254/2011. Cited on page 8.

[TITW05] Akihiro Taguchi, Atsushi Iriboshi, Satoshi Taoka, and Toshimasa Watanabe. Siphon-trap-based algorithms for efficiently computing petri net invariants. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E88-A:964–971, April 2005. Cited on page 147.

[Tur49] Alan Mathison Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Mathematical Laboratory, Cambridge, 1949. Cited on page 4.

[Val78a] Rüdiger Valk. On the computational power of extended petri nets. In Józef Winkowski, editor, *Proceedings of the 7th International Symposium on Mathematical Foundations of Computer Science (MFCS'78)*, volume 64 of *Lecture Notes*

in Computer Science, pages 526–535. Springer Berlin Heidelberg, 1978. Cited on page 38.

[Val78b] Rüdiger Valk. Self-modifying nets, a natural extension of petri nets. In Giorgio Ausiello and Corrado Böhm, editors, *Proceedings of the 5th International Colloquium on Automata, Languages and Programming (ICALP'78)*, volume 62 of *Lecture Notes in Computer Science*, pages 464–476. Springer Berlin Heidelberg, 1978. Cited on page 38.

[Val90] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 – Selected Papers of the Proceedings of the 10th International Conference on the Applications and Theory of Petri Nets (ATPN'89)*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer Berlin Heidelberg, 1990. Cited on page 147.

[Van04] Laurent Van Begin. *Efficient Verification of Counting Abstractions for Parametric Systems.* PhD thesis, Université Libre de Bruxelles, Faculté des Sciences, Département d'Informatique, 2004. Cited on pages 246 and 256.

[VH12] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. In Serge Haddad and Lucia Pomello, editors, *Proceedings of the 33rd International Conference on the Application and Theory of Petri Nets and Concurrency (ATPN'12)*, volume 7347 of *Lecture Notes in Computer Science*, pages 208–227. Springer Berlin Heidelberg, 2012. Cited on page 6.

[WKP09] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. Transition-based directed model checking. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 186–200. Springer Berlin Heidelberg, 2009. Cited on page 156.

[YD98] Chiang Han Yang and David L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th annual Design Automation Conference (DAC'98)*, pages 599–604, New York, NY, USA, 1998. ACM. Cited on page 156.

[ZL95] Denis Zampuniéris and Baudouin Le Charlier. Efficient handling of large sets of tuples with sharing trees. In *Proceedings of the IEEE Data Compression Conference (DCC'95)*, page 428. IEEE Computer Society, Los Alamitos, CA, USA, 1995. Cited on page 160.

Index

Symbols
$\perp$-child....................173
$dist(X, Y)$ .... *see* distance
$\downarrow X$ .... *see* downward-closed set
$lbl_y(x)$ .... *see* labelling function
$\ominus$ .... *see* monus
pb .... *see* pred-pasis
pre .... *see* predecessor
$\preceq$ .... *see* quasi ordering
$\top$-child....................173
$\uparrow X$ ...... *see* upward-closed set
$\preceq$ .... *see* well-quasi ordering
$wit_x(y)$ .... *see* witness

Abbreviations
A^* .... *see* A-star
BDD .... *see* binary decision diagram
BF .... *see* best-first search
BFS ... *see* breadth-first search
BR .. *see* backward reachability analysis
BST .... *see* binary search tree
CFG *see* context-free grammar
CST . *see* covering sharing tree
DCS .. *see* downward-closed set
DFS ..... *see* depth-first search
EEC . *see* expand, enlarge, and check
EXPSPACE ... *see* complexity, exponential space
FIFO ..... *see* first in, first out
HTS *see* holonic transportation system
IST ... *see* interval sharing tree
JVM . *see* Java virtual machine
LCS .. *see* lossy channel system
MDD *see* multi-valued decision diagram
NP .... *see* complexity, non-polynomial
P .. *see* complexity, polynomial
PN .... *see* Petri net
PNT .... *see* Petri net, transfer

POR .... *see* partial-order reduction
PORPA .... *see* combined optimization
PSPACE ...... *see* complexity, polynomial space
PST .. *see* powerset search tree
PW .... *see* correctness, partial
QO .... *see* quasi ordering
SSC .... *see* search space construction
TTS ..... *see* thread transition system
TW ...... *see* correctness, total
UCS ..... *see* upward-closed set
UML ..... *see* unified modeling language
WQO .. *see* well-quasi ordering
WSLTS .... *see* well-structured transition system, labelled
WSTS ..... *see* well-structured transition system

A

abstraction *see* data abstraction
accelerated predecessor 132, 133
acceleration 131
acceleration candidate . 132, 214
Ackermann function 208
alternating bit protocol 41
anti-chain **18**, 160
arc **12**
  inhibitor 36
  weight 12, 37
assignment axiom **44**, 57
assumption 45

B

backward acceleration **133**
backward coverability analysis *see* backward reachability analysis
backward reachability analysis **30**, 226
basis .... *see* upward-closed set, basis
  finite *see* upward-closed set, finite basis, *see* downward-closed set, finite basis
  minimal 69
BFC 215
binary decision diagram ... 160, 220
binary search tree ..... *see* tree, search
binary tree ..... *see* tree, binary
block 165
breadth 136

C

candidate sequence 214
channel .... *see* lossy channel system, channel
characteristic set **172**
child node .. *see* tree, child node
combined optimization **153**
composition rule **44**
conditional rule **44**
configuration . *see* lossy channel system, configuration
consequence rule **45**, 57, 112
consistency **34**
context-free grammar 195

control state . . *see* lossy channel system, control state
correctness **43, 44**
  formula 43
  partial **44**
  total **45**
  while programs **43**
coverability **20**
  graph 15
coverability problem **24**
  labelled **35**
coverability relation **23**
  labelled **34**
covering . . . . *see* coverability
covering predecessor . . . . *see* predecessor, covering
covering sharing tree . . 160, 220
critical section 36

D

data abstraction 192, 201
data structure 160
default value 76, 90, 91
delegate buffer program255–264
depth 136
directed search . . . . . . *see* guided search
distance **82**, 82
distance reducing 82
downward-closed set **29**
  finite basis **29**

E

edge . . . . *see* tree, edge
enabled 13
equality condition 166, 167
equivalence class **165**, 165
equivalence relation
  induced **165**
expand, enlarge, and check . 220
EXPSPACE-complete 36

F

factorial 48
firing 13
first in, first out 40
flag interface 194
foreach loop 71
Framework 216
function assignment **76**, 90, 168
function assignment axiom . . **90**

G

generics 194
guided search 85, **155**

H

height . . . . *see* tree, height
hierarchy 165, 185
high-level 125
Hoare proof . . . . . *see* correctness
holonic transportation system 219, 239–246
hybrid system 220

I

identity 24
image-finite 176
indicator function **165**, 192, 199
indicator value **165**
inner node . *see* tree, inner node
interface 193
interval sharing tree 160
invariant 45, 147
inverse monotonicity 83

J

Java annotation 196
Java assertions 205
Java virtual machine . . 205, 216

K

Kanban production system 250–255

L

labelling function **81**
leaf node . . . . *see* tree, leaf node
local operation *see* lossy channel system, local operation
loop II rule **45**
loop rule **44**
lossiness 40
lossy channel system **40**
- channel 40
- configuration 40, 166
- control state 40
- local operation 40
- message 40
- operation 40
- pred-basis **151**
- receive operation 40
- send operation 40
- update 40

low-level 125

M

marking . *see* Petri net, marking
merge operation **181**
message . . . . . . *see* lossy channel system, message
method 193
minimize **79**
minimize axiom **88**, 90
MIST2 215
monotone dual-reference program 215
monus **141**
multi-layered architecture . . 193
multi-valued decision diagram 160
multiset 13
mutual exclusion 36

N

necessary condition 164
node . . . . *see* tree, node
non-primitive recursive complexity 38, 41
nondeterministic assignment . 72

O

object 193
ω-marking 15
operation . . . . . *see* lossy channel system, operation, 40
optimized predecessor 74, 82, **83**
order
- lexicographical **63**, **110**

out-degree 136

P

parent node . . . . *see* tree, parent node
partial-order reduction 15, **147**, 147
Petri net **12**, **36**
- marking **13**
- pred-basis **141**
- timed 36
- transfer **37**, 256

Petruchio/BW . . . 215
π-calculus . . . 8, 240
place . . . **12**
  unbounded . . . 15
plateau . . . 132
PNCSA protocol . . . 246–250
polyhedron . . . 220
postcondition . . . 43
postset . . . 13, 157
powerset search tree . . . 172, 192
  full . . . **172**
  relaxed . . . **176**
precondition . . . 43
pred-basis . . . **29**
  lossy channel system . . . *see* lossy channel system, pred-basis
  Petri net . . . *see* Petri net, pred-basis
predecessor . . . **24**
  covering . **23, 24**, 134, 139, 205
predecessor basis . *see* pred-basis
preset . . . 13
program correctness . . . *see* correctness
proof outline . . . 55
  partial correctness . . . **47**
  total correctness . . . **47**
proof system . . . *see* correctness
pruning . . 15, 143, **144**, 144, 146

Q

quasi ordering . . . **16**
quotient set . . . 165

R

rapid prototyping . . . 223
receive operation . . . *see* lossy channel system, receive operation
rooted tree . . . *see* tree, binary

S

search space construction . . . 208
search strategy . . . *see* guided search
search tree . . . *see* tree, search
select . . . **30**
select axiom . . . **61**
send operation *see* lossy channel system, send operation
sharing tree . . . 160
  covering . . . 160
  interval . . . 160
shuffle . . . 211
skip axiom . . . **44**
stabilization . . . *see* well-quasi ordering, stabilization
stress test . . . 205
subalgorithm . . . 88
subset condition . . . 166, 170
subword ordering . . . 41
support . . . 171
syntactic distance . . . 156
syntactic weight . . . 157

T

target set widening . . . 215, 220
target state . . . 23
theorem . . . 46
thread transition system . . . 215
total order condition . . 166, 168
transition
  Petri net . . . 12
tree

- binary **171**
- child node **171**
- edge **171**
- extension **180**
- height **171**
- inner node **171**
- leaf node **171**
- node **171**
- parent node **171**
- powerset search . . . . *see* powerset search tree
- rooted . . . . . *see* tree, binary
- search 172

U

unbounded . . . . *see* place, unbounded
unified modeling language . . 199
update operation . . . . . . *see* lossy channel system, update
upward-closed set **20**
- basis **20**
- finite basis **22**, 160

upward-compatibility . . . **19**, **34**
- transitivity **20**

W

weight function . *see* arc, weight
well-foundedness **18**
well-quasi ordering **17**
- anti-chain **18**
- stabilization **27**
- subsequence **18**
- well-foundedness **18**

well-structured transition system **19**
- labelled **34**

while program
- syntax **43**

witness **84**
- trivial **84**

Technical Reports

Fakultät II, Department für Informatik, Universitä Oldenburg,
Postfach 2503, 26111 Oldenburg, Germany

1/87 A. Viereck: „Klassifikationen, Konzepte und Modelle für den Mensch-Rechner-Dialog“ (Dissertation)

2/87 A. Schwill: „Forbidden subgraphs and reduction systems: A comparison“

3/87 J. Kämper: „Non-uniform proof systems: A new framework to describe non-uniform and probabilistic complexity classes“

1/88 K. Ambos-Spies, H. Fleischhack, H. Huwig: „Diagonalizing over deterministic polynomial time“

2/88 A. Schwill: „Shortest edge-disjoint paths in geodetically connected graphs“

3/88 V. Claus, U. Lichtblau (Hrsg.): „1. Tagung zur Küsten-Informatik“

1/89 U. van der Valk: „Einige Entscheidbarkeits- und Unentscheidbarkeitsresultate für Klasse von S/T-Netzen unter Maximum Firing Strategie und unter Prioritätenstrategien“

2/89 J. Kämper: „Strukturelle Untersuchungen im Umfeld der Komplexitätsklassen P und NP unter besonderer Berücksichtigung nichtuniformer, probabilistischer und disjunktiv selbstreduzierender Algorithmen“ (Dissertation)

3/89 J. Kämper: „Nondeterministic oracle Turing machines with maximal computation paths“

1/90 A. Schwill: „Shortest edge-disjoint paths in graphs“ (Dissertation)

2/90 K.R. Apt, E.-R. Olderog: „Using transformations to verify parallel programs“

3/90 U. Lichtblau: „Flußgraphgrammatiken“ (Dissertation)

4/90 K.R. Apt, E.-R. Olderog: „Introduction to program verification“

5/90 H. Jasper: „Datenbankunterstützung für Prolog-Programmierumgebungen“ (Dissertation)

1/91 F. Korf: „Net-based efficient simulation of AADL specifications“

2/91 S.V. Krishnan, C. Pandu Rangan, A. Schwill, S. Seshadri: „Two disjoint paths in chordal graphs“

3/91 H. Eirund: „Modellierung und Manipulation multimedialer Dokumente“ (Dissertation)

4/91 G. Schreiber: „Ein funktionaler Äquivalenzbegriff für den hierarchischen Entwurf von Netzen“

1/92 A. Viereck (Hrsg.): „Ergebnisse der 11. Arbeitstagung, Mensch-Maschine Kommunikation“

2/92 P. Gorny, U. Daldrup, H. Schwab: „Zwischenbilanz: Menschengerechte Gestaltung von Software“

3/92 E.-R. Olderog, St. Rössig, J. Sander, M. Schenke: „ProCoS at Oldenburg: The Interface between Specification Language and occam-like Programming Language“

4/92 F. Korf: „Synthesis of VHDL Test Environments form Temporal Logic Specifications“

5/92 W. Kowalk: „Konstruktorentechnik: Neue Methoden zur Mengenrechnung, Logikrechnung und Intervallrechnung“

1/93 Ch. Dietz, G. Schreiber: „Eine Termdarstellung für S/T-Netze“

2/93 J. Sauer: „Wissensbasiertes Lösen von Ablaufplanungsproblemen durch explizite Heuristiken“

3/93 M. Sonnenschein, U. Lichtblau (Hrsg.): „6. Kolloquium der Arbeitsgruppe Informatik-Systeme“

4/93 H. Fleischhack, U. Lichtblau, M. Sonnenschein, R. Wieting: „Generische Definition hierarchischer zeitbeschrifteter höherer Petrinetze“

5/93 F. Köster, L. Twele, R. Wieting, W. Ziegler: „Fallbeispiele zur Modellierung mit THORNetzen“

1/94 R. Götze: „Dialogmodellierung für multimediale Benutzerschnittstellen"

2/94 B. Müller: „PPO-Eine objektorientierte Prolog-Erweiterung zur Entwicklung wissensbasierter Anwendungssysteme"

3/94 W. Damm/A. Mikschl: „Projekt Entwurf und Implementierung eines multi-threaded RISC-Prozessors"

4/94 S. Rössig: „A Transformational Approach to the Design of Communicating Systems" (Dissertation)

5/94 G. Schreiber: „Funktionale Äquivalenz von Petri-Netzen" (Dissertation)

1/95 A. Gronewold, H. Fleischhack: „Language Preserving Reductions of Safe Petri-Nets"

2/95 H. Reineke: „Struktur und Verhalten von verteilten endlichen Automaten" (Dissertation)

3/95 H. Behrends: „Beschreibung ereignisgesteuerter Aktivitäten in datenbankgestützten Informationssystemen" (Dissertation)

4/95 U. M. Levens: „Computerunterstütztes Modellieren von Musikstücken mit Petri-Netzen: Das Mailänder Konzept"

1/96 M. Burke: „FDDI und ATM in multimedialen Anwendungsumgebungen" (Dissertation)

2/96 I. Pitschke: „Interaktive Rekonstruktion geometrischer Modelle aus digitalen Bildern" (Dissertation)

1/97 L. Bölke: „Ein akustischer Interaktionsraum für blinde Rechnerbenutzer" (Dissertation)

2/97 S. Schöf: „Verteilte Simulation höherer Petrinetze" (Dissertation)

1/98 S. Kleuker: „Inkrementelle Entwicklung von verifizierten Spezifikationen für verteilte Systeme" (Dissertation)

2/98 J. Bohn: „Mechanical Support and Validation of a Design Calculus for Communicating Systems by a Logic-Based Proof System" (Dissertation)

3/98 L. Köhler: „Fuzzy Geometrie und Anwendungen in der medizinischen Bildverarbeitung" (Dissertation)

4/98 J. Helbig: „Linking Visual Formalisms: A Compositional Proof System for Statecharts Based on Symbolic Timing Diagrams" (Dissertation)

5/98 G. Stiege: „Edge Partitions in Undirected Graphs"

6/98 A. Gerns: „Entwicklung und Bewertung von Objektmigrationsstrategien für verteilte Umgebungen"

7/98 M. Stadler: „Abstrakte Rechnernetzmodelle als Grundlage einer umfassenden Automatisierung des Netzmanagements – Konzepte und Sprachen zu ihrer Umsetzung“ (Dissertation)

8/98 M.-S. Steiner: „Lastverteilung in heterogenen Systemen“

9/98 Clemens Otte: „Fuzzy-Prototyp-Klassifikatoren und deren Anwendung zur automatischen Merkmalsselektion“

1/99 Juliane Vorndamme: „Die Auswirkungen rechtlicher Verpflichtungen auf die Softwareentwicklung“

2/99 E. Best/K.M. Richter: „Relational Semantics Revisited“

3/99 J. S. Lie: „Einsatz von Objektmigrationssystemen zur Leistungssteigerung in verteilten Systemen“

4/99 „Zwei-Jahresbericht des Fachbereichs Informatik, 01.10.96 – 30.09.98“

5/99 Ingo Stierand, Olaf Maibaum, Björn Briel, Günther Stiege: „Cassandra – Generierung, Analyse und Simulation von eingebetteten Multiprozessor-Echtzeitsystemen“

6/99 Gunnar Wittich: „Ein problemorientierter Ansatz zum Nachweis von Realzeiteigenschaften eingebetteter Systeme“

7/99 Annegret Habel, Jürgen Müller, Detlef Plump: „Double-Pushout Graph Transformation Revisited“

8/99 Ingo Stierand: „Eine Konfigurationssprache zur Erstellung von Ambrosia/MP-Systemen“

9/99 Igor V. Tarasyuk: „Equivalences for Concurrent and Distributed Systems“

10/99 Eike Best, Alexander Lavrov: „Generalised Composition Operations for High-Level Petri-Nets“

11/99 Alexander Lavrov: „Enhancing Mixed Nonlinear Optimization: A Hybrid Approach“

12/99 Alexander Lavrov: „Hybrid Techniques in Discrete-Event System Modelling and Control: some Examples“

13/99 Eike Best, Raymond Devillers, Maciej Koutny: „Recursion and Petri Nets“

14/99 Eike Best, Raymond Devillers, Maciej Koutny: „The Box Algebra = Petri Nets + Process Expressions“

15/99 Eike Best, Harro Wimmel: „Reducing k-safe Petri Nets to Pomset-equivalent 1-safe Petri Nets“

16/99 Udo Brockmeyer: „Verifikation von STATEMATE Designs“ (Dissertation)

1/00 Henning Dierks: „Specification and Verification of Polling Real-Time Systems“ (Dissertation)

2/00 Clemens Fischer: „Combination and Implementation of Processes and Data: from CSP-OZ to Java“ (Dissertation)

3/00 Cheryl Kleuker: „Constraint Diagrams“ (Dissertation)

4/00 Thomas Thielke: „Linear-algebraische Methoden zur Beschreibung, Verfeinerung und Analyse gefärbter Petrinetze“ (Dissertation)

1/01 Günther Stiege: „Higher Decomposition in Undirected Graphs“ (Bericht)

2/01 Ute Vogel: „Zwei-Jahres-Bericht des Fachbereichs Informatik, 01.10.1998 – 30.09.2000“

3/01 Josef Tapken: „Model-Checking of Duration Calculus Specifications“ (Dissertation)

4/01 Björn Briel: „Analyse eingebetteter Systeme mittels verteilter Simulation“ (Dissertation)

5/01 Günther Stiege: „Standard Decomposition and Periodicity of Digraphs“ (Bericht)

6/01 Ingo Stierand: „Ambrosia/MP – Ein Echtzeitbetriebssystem für eingebettete Mehrprozessorsysteme“ (Dissertation)

1/02 Giorgio Busatto, Annegret Habel: „Improving the Quality of Hypertexts Using Graph Transformation“ (Bericht)

2/02 Giorgio Busatto: „Modeling Hyperweb Dynamics through Hierarchical Graph Transformation“ (Bericht)

3/02 Giorgio Busatto: „An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation“ (Dissertation)

4/02 Laila Kabous: „An Object Oriented Design methodology for hard real Time Systems: The OOHARTS approach“ (Dissertation)

1/03 Ute Vogel: „2-Jahres-Bericht, 01.10.2000 – 30.09.2002“

2/03 Olaf Maibaum: „Bestimmung symbolischer Laufzeiten in eingebetteten Echtzeitsystemen“ (Dissertation)

3/03 Günther Stiege, Ingo Stierand: „Connectedness-Based Hierarchical Decomposition of Undirected Graphs“ (Bericht)

4/03 Willi Hasselbring, Susanne Petersen: „Standards für die medizinische Kommunikation und Dokumentation“ (Bericht)

5/03 Andreas Möller: „Eine virtuelle Maschine für Graphprogramme" (Bericht)

6/03 Tom Bienmüller: „Reducing Complexity for the Verification of Statemate Designs" (Bericht)

7/03 Sandra Steinert: „Graph Programs for Graph Algorithms" (Bericht)

8/03 Jochen Klose: „Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior" (Dissertation)

1/04 Jens Oehlerking: „Transformation of Edmonds' Maximum Matching Algorithm into a Graph Program" (Bericht)

2/04 Sergej Alekseev: „Dienste Intelligenter Netze Graphentheoretische Methoden in der Kontrollflussanalyse" (Bericht)

3/04 Giorgio Busatto: „GraJ: A System für Executing Graph Programs in Java" (Bericht)

1/05 Sergej Alekseev and Johannes Wust: „Graph Theoretical Methods in the Control Flow Analysis of Object Oriented Real Time Software" (Bericht)

2/05 Ute Vogel: „2-Jahres-Bericht, 01.10.2002 – 30.09.2004"

3/05 Igor V. Tarasyuk: „Discrete time stochastic Petri box calculus" (Bericht)

1/06 Henning Dierks: „Time, Abstraction and Heuristics" (Habilitation)

2/06 Li Sek Su: „Full-Output Siphons and Deadlock-Freeness for Free Choice Petri Nets" (Bericht)

3/06 Timo Warns: „Solving Consensus Using Structural Failure Models" (Bericht)

4/06 Sergej Alekseev: „Graphentheoretische Methoden in der Ablaufanalyse objektorientierter Anwendungen" (Dissertation)

5/06 Li Sek Su: „Some Considerations on the Foundation of NP-Completeness Theory*" (Bericht)

6/06 Li Sek Su: „Semitraps and Deadlock-Freeness for Reduced Asymmetric Choice Nets" (Bericht)

7/06 Li Sek Su: „Algorithms of computing the Deadlock Markings Sets for Petri Nets" (Bericht)

8/06 Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink: „Weakest Preconditions for High-Level Programs (Long Version)" (Bericht)

9/06 Jochen Hoenicke: „Combination of Processes, Data, and Time" (Dissertation)

10/06 Steffen Becker, Marco Boscovic, Abhishek Dhama, Simon Giesecke, Jens Happe, Wilhelm Hasselbring, Heiko Koziolek, Henrik Lipskoch, Roland Meyer, Margarethe Muhle, Alexandra Paul, Jan Ploski, Matthias Rohr, Mani Swaminathan, Timo Warns, Daniel Winteler: „Trustworthy Software Systems: A Discussion of Basic Concepts and Terminology“ (Bericht)

11/06 Christian Zuckschwerdt: „Ein System zur Transformation von Konsistenz- in Anwendungsbedingungen“ (Bericht)

01/07 Andreas Schäfer: „Specification and Verification of Mobile Real-Time Systems“ (Dissertation)

02/07 Günther Stiege: „General Graphs“ (Bericht)

03/07 Wolfgang Kowalk: „Integralrechnung“ (Bericht)

04/07 Karl Azab, Karl-Heinz Pennemann: „Type Checking C++ Template Instantiation by Graph Programs“ (Bericht)

01/08 Roland Meyer: „On depth and breath in the Pi-Calculus“ (Bericht)

02/08 Ingo Brückner: „Slicing Integrated Formal Specifications for Verification“ (Dissertation)

03/08 Ute Vogel: „2-Jahres-Bericht, 01.10.2004 – 30.09.2006“ (Bericht)

04/08 Günther Stiege: „Summierbare Familien“ (Bericht)

05/08 Igor V. Tarasyuk: „Investigating equivalence relations in dtsPBC“ (Bericht)

01/09 Elke Wilkeit: „2-Jahres-Bericht, 01.10.2006 – 30.09.2008“ (Bericht)

02/09 Roland Meyer: „Structural Stationarity in the pi-Calculus“ (Dissertation)

03/09 InformatikerInnen des Moduls Soft Skills: „E-Book Soft Skills 2008“ (Bericht)

04/09 Eike Best: „Separability in Persistent Petri Nets“ (Bericht)

01/10 Igor V. Tarasyuk: „Equivalence relations for behaviour-preserving reduction and modular performance evaluation in dtsPBC“ (Bericht)

02/10 Roman Dubtsov: „Timed Transition Systems with Independence and Marked Scott Domains: an Adjunction“ (Bericht)

01/11 Elena S. Oshevskaya: „Matching Equivalences on Higher Dimensional Automata Models“ (Bericht)

02/11 Elke Wilkeit: „2-Jahres-Bericht, 01.10.2008 – 30.09.2010“ (Bericht)

03/11 Johannes Faber: „Verification Architectures for Complex Real-Time Systems“ (Dissertation)

04/11 Igor V. Tarasyuk: „Equivalences for modular performance analysis in dtsPBC“ (Bericht)

01/12 Irina Virbitskaite, Nataliya Gribovskaya, Eike Best: „Some Evidence on the Consistency of Categorical Semantics for Timed Interleaving Behaviours“ (Bericht)

1/12 Günther Stiege: „Playing with Knuth's words.dat“ (Bericht)

02/12 Günther Stiege: „Flowerfree Finding of Maximum Matchings“ (Bericht)

01/13 Wolfgang Kowalk: „RunSort – Ein effizienter Sortieralgorithmus“ (Bericht)

02/13 Günther Stiege: „Cliques and Graphs of Type WORDS“ (Bericht)

03/13 Jan-David Quesel: „Similarity, Logic, and Games – Bridging Modeling Layers of Hybrid Systems“ (Dissertation)

04/13 Günther Stiege: „Vertex Coloring the Complements of Knuth Graphs“ (Bericht)