Techniques and Tools for the Verification of Concurrent Systems

Hristina Palikareva

Linacre College

A thesis submitted to the
Department of Computer Science of
University of Oxford
for the degree of
Doctor of Philosophy

Michaelmas Term 2012

Abstract

Model checking is an automatic formal verification technique for establishing correctness of systems. It has been widely used in industry for analysing and verifying complex safety-critical systems in application domains such as avionics, medicine and computer security, where manual testing is infeasible and even minor errors could have dire consequences.

In our increasingly parallelised world, concurrency has become pivotal and seamlessly woven within programming paradigms, however, extremely challenging when it comes to modelling and establishing correctness of intended behaviour. Tools for model checking concurrent systems face severe limitations due to scalability problems arising from the need to examine all possible interleavings (schedules) of executions of parallel components. Moreover, concurrency poses additional challenges to model checking, giving rise to phenomena such as nondeterminism, deadlock, livelock, etc.

In this thesis we focus on adapting and developing novel model-checking techniques for concurrent systems in the setting of the process algebra CSP and its primary model checker FDR. CSP allows for a compact modelling and precise analysis of event-based concurrency, grounded on synchronous message passing as a fundamental mechanism of inter-component communication. In particular, we investigate techniques based on symbolic model checking, static analysis and abstraction, all of them exploiting the compositionality inherent in CSP and targeting to increase the scale of systems that can be tractably analysed.

Firstly, we investigate symbolic model-checking techniques based on Boolean satisfiability (SAT), which we adapt for the traces model of CSP. We tailor bounded model checking (BMC), that can be used for bug detection, and temporal k-induction, which aims at establishing inductiveness of properties and is capable of both bug finding and establishing the correctness of systems.

Secondly, we propose a static analysis framework for establishing livelock freedom of CSP processes, with lessons for other concurrent formalisms. As opposed to traditional exhaustive state-space exploration, our framework employs a system of rules on the syntax of a process to calculate a sound approximation of its fair/co-fair sets of events. The rules either safely classify a process as livelock-free or report inconclusiveness, thereby trading accuracy for speed.

Finally, we develop a series of abstraction/refinement schemes for the traces, stable-failures and failures-divergences models of CSP and embed them into a fully automated and compositional CEGAR framework.

For each of those techniques we present an implementation and an experimental evaluation on a set of CSP benchmarks.

Acknowledgements

First and foremost, I would like to thank my supervisors Bill Roscoe and Joël Ouaknine for all their guidance, support, enthusiasm and inspiration throughout the years—it has been a truly remarkable experience in many ways and I have learned from them so much. I am also very grateful to James Worrell for all his support and advice, working with him has been thoroughly rewarding and enjoyable. I would like to also express my gratitude to my examiners Daniel Kröning and Steve Schneider for their profound analysis of my work and for the stimulating discussion during my viva. I would like to thank the Department of Computer Science at the University of Oxford, and especially the Verification Group, for the wonderful and inspiring environment—it has been a true honour and joy to be a part of this academic family. I am also extremely grateful to EPSRC for fully funding my doctoral studies and to Linacre College for all the support throughout the years. And finally, I would like to say huge thanks to all my family and friends in Bulgaria, Oxford and all around the world.

Contents

1 Introduction **1**
- 1.1 Contributions of the Thesis . . . 7
  - 1.1.1 SAT-Based Trace Refinement Checking. . . . 7
  - 1.1.2 Static Analysis for Establishing Livelock Freedom. . . . 9
  - 1.1.3 Abstraction Schemes and CEGAR Framework for CSP and FDR. . 10
- 1.2 Published Work . . . 11
- 1.3 Layout of the Thesis . . . 12

2 Background Material **14**
- 2.1 Formalisms and Notations . . . 14
  - 2.1.1 Alphabets, Words and Languages . . . 14
  - 2.1.2 Labelled Transition Systems . . . 15
  - 2.1.3 Relations . . . 16
    - 2.1.3.1 Partial Orders and Lattices . . . 16
    - 2.1.3.2 Equivalence Relations and Partitions . . . 19
  - 2.1.4 Propositional Logic . . . 20
- 2.2 CSP . . . 22
  - 2.2.1 Syntax . . . 23
  - 2.2.2 Example: Milner's Scheduler . . . 26
  - 2.2.3 A Hierarchy of Denotational Models . . . 28
    - 2.2.3.1 The Traces Model. . . . 30
    - 2.2.3.2 The Stable-Failures Model. . . . 32
    - 2.2.3.3 The Failures-Divergences Model. . . . 34
    - 2.2.3.4 The Hierarchy . . . 36
    - 2.2.3.5 Calculating Denotational Values . . . 37
  - 2.2.4 Operational Semantics . . . 37
    - 2.2.4.1 SOS Rules . . . 38
    - 2.2.4.2 Deriving Behaviours from Operational Semantics . . . 40
    - 2.2.4.3 Bisimulation Orders . . . 41
- 2.3 FDR . . . 44
  - 2.3.1 The Two-Level Operational Semantics . . . 44
    - 2.3.1.1 Combinator-Style Operational Semantics . . . 44
    - 2.3.1.2 Supercombinator-Style Operational Semantics . . . 47
  - 2.3.2 Refinement Checking . . . 51

2.3.3 Abstraction and Compression Techniques . . . 54
2.3.3.1 Semantic-Preserving Transformations on Transition Systems 54
2.3.3.2 Hierarchical Compression . . . 55

3 SAT-Solving Techniques for CSP Refinement Checking 58
3.1 Introduction . . . 58
3.2 Foundations . . . 62
3.2.1 SAT Essentials . . . 62
3.2.2 Bounded Model Checking . . . 63
3.2.3 From Bounded to Complete Model Checking . . . 65
3.2.3.1 Temporal k-induction . . . 65
3.3 Bounded Trace Refinement Framework . . . 67
3.3.1 Challenges . . . 67
3.3.2 Watchdog Refinement-Checking Algorithm . . . 68
3.3.2.1 Preprocessing Phase Using FDR . . . 69
3.3.2.2 Watchdog Bounded Refinement-Checking Algorithm . . . 69
3.4 Boolean Encoding of CSP Processes . . . 71
3.4.1 Encoding Sets . . . 72
3.4.2 Encoding a Sequential Process . . . 73
3.4.3 Encoding a Concurrent System . . . 74
3.5 Implementation Details . . . 77
3.5.1 BMC . . . 77
3.5.2 k-induction . . . 78
3.5.3 SAT . . . 78
3.6 Experimental Results . . . 79
3.6.1 Comparison Tools . . . 79
3.6.2 Test Cases . . . 81
3.6.2.1 Instances with Counterexamples . . . 81
3.6.2.2 Instances without Counterexamples . . . 85
3.6.2.3 Discussion . . . 87
3.7 Conclusion and Future Work . . . 87
3.7.1 Interpolation Techniques . . . 88
3.7.2 IC3 and PDR Frameworks . . . 90

4 Static Analysis for Livelock Detection 92
4.1 Introduction . . . 92
4.1.1 Sources of Livelock . . . 93
4.1.2 Complexity of Establishing Livelock Freedom . . . 93
4.1.3 Our Proposal: Static Analysis . . . 94
4.1.4 Related Work . . . 96
4.2 Case Study: the Alternating Bit Protocol . . . 97
4.2.1 Description and Implementation . . . 97
4.2.2 An Abstracted Version of the ABP . . . 100
4.3 Syntax and Conventions . . . 101
4.4 Operational and Denotational Semantics . . . 102

4.4.1 Operational Semantics 102
4.4.2 Denotational Semantics 102
4.4.2.1 The Model $\mathcal{T}^{\Downarrow}$ 102
4.4.2.2 Reasoning about Infinite Traces 105
4.4.2.3 Handling Recursion 106
4.5 Processes and Metric Spaces 106
4.5.1 Metric Spaces 106
4.5.2 Introducing a New Family of Metrics 108
4.5.3 Nonexpansiveness of CSP Operators 111
4.6 Static Livelock Analysis: General Framework 112
4.6.1 Nonexpansiveness and Guardedness 113
4.6.2 Fair Sets and Hiding 118
4.6.3 Examples and Discussion 120
4.7 Static Livelock Analysis for Structurally Finite-State Processes 122
4.7.1 Goal 124
4.7.2 Handling Atomic $\overline{\mathsf{SFS}}$ Processes 125
4.7.2.1 Optimisations Considering Strongly Connected Components 126
4.7.3 Compositional Rules for Compound $\overline{\mathsf{SFS}}$ Processes 126
4.7.4 Examples and Discussion 128
4.7.4.1 The Abstracted Version of the ABP 128
4.7.4.2 Static Livelock Analysis Algorithm 130
4.7.4.3 Notions of Fairness 131
4.8 Implementation Details and Symbolic Encoding 132
4.8.1 The $\overline{\mathsf{SFS}}$ Framework 133
4.8.1.1 Computing Fair/Co-Fair Sets for Atomic $\overline{\mathsf{SFS}}$ Processes . . 133
4.8.1.2 Encoding Compositional Rules 136
4.8.2 The General Framework 136
4.9 Experimental Results 138
4.9.1 Infinite-State Processes 139
4.9.2 Structurally Finite-State Processes 140
4.10 Conclusion and Future Work 143

5 Abstraction Schemes and CEGAR Framework for CSP and FDR 146
5.1 Introduction 146
5.1.1 Abstraction 146
5.1.2 Counterexample-Guided Abstraction Refinement 148
5.1.3 Introducing a CEGAR Framework for CSP and FDR 148
5.1.4 Related Work 149
5.2 General Skeleton of the CEGAR Framework for CSP and FDR 151
5.3 Abstraction Schemes for CSP 156
5.3.1 Requirements 156
5.3.2 An Abstraction for the Traces Model 157
5.3.3 A Collection of Abstractions for the Stable-Failures Model 159
5.3.4 Extensions for the Failures-Divergences Model 161
5.4 Refinement Checking and Counterexample Validation 162

5.4.1 Types of Counterexample Behaviours 163
5.4.2 Counterexample Validation of Finite Traces, Minimal Acceptances and Divergences 164
5.4.3 Counterexample Validation of Lasso Traces 166
5.5 Abstraction Refinement 168
5.5.1 Simulation of Abstract Executions 170
5.5.2 Abstraction-Refinement Strategies for Spurious Finite Traces 171
5.5.3 Abstraction-Refinement Strategies for Spurious Minimal Acceptances 173
5.5.4 Abstraction-Refinement Strategies for Spurious Divergences 176
5.5.5 Abstraction-Refinement Strategies for Spurious Lasso Traces 177
5.6 Implementation Details 178
5.6.1 Aggressive vs. Lazy Strategies 179
5.7 Experimental Results 180
5.8 Conclusion and Future Work 185

6 Conclusions and Future Work **190**
6.1 Summary and Evaluation 190
6.2 Future Work 196

A Proofs **200**
A.1 Proofs for Section 4.4.2.2 200
A.2 Proofs for Section 4.5.2 205
A.3 Proofs for Section 4.5.3 207
A.4 Proofs for Section 4.6 215
A.5 Proofs for Section 4.7 232

List of Figures

2.1 Tseitin encoding rules . . . 22
2.2 Tseitin encoding rules: multiple disjunctions and conjunctions . . . 22
2.3 Milner's scheduler for $N = 5$ . . . 27
2.4 CSP syntax: Milner's scheduler . . . 27
2.5 Order-theoretic structure of denotational models . . . 30
2.6 The model $\mathcal{T}$: inductive rules for calculating traces . . . 31
2.7 Interleaving operator on traces . . . 32
2.8 The model $\mathcal{F}$: inductive rules for calculating stable failures . . . 33
2.9 The model $\mathcal{N}$: inductive rules for calculating divergences . . . 36
2.10 Operational representation of CSP processes . . . 37
2.11 Process tree for $\mathit{Scheduler} \setminus \{\mid b \mid\}$ . . . 50
2.12 Operational semantics of a high-level process . . . 51

3.1 Encoding paths of length k . . . 65
3.2 Forward k-induction ingredients . . . 66
3.3 Backward k-induction ingredients . . . 67

4.1 Livelock . . . 92
4.2 The alternating bit protocol . . . 98
4.3 ABP: the sender process . . . 98
4.4 ABP: the receiver process . . . 99
4.5 ABP: the lossy media . . . 99
4.6 ABP: an abstracted version . . . 101
4.7 The model $\mathcal{T}^{\Downarrow}$: inductive rules for calculating traces . . . 104
4.8 The model $\mathcal{T}^{\Downarrow}$: inductive rules for calculating divergences . . . 104
4.9 Contractive and nonexpansive maps . . . 108
4.10 Collecting sets of pairs of metrics . . . 115
4.11 Nonexpansive sets . . . 116
4.12 Guard sets . . . 117
4.13 Contractive sets . . . 118
4.14 Fair sets . . . 119
4.15 An infinite buffer . . . 120
4.16 Abstracted ABP: transition systems . . . 121
4.17 Precision layers . . . 123
4.18 Fair/co-fair sets . . . 127
4.19 δ-bit . . . 127

4.20 Abstracted version of the ABP 128
4.21 Static livelock analysis: a false positive 130
4.22 Calculating fair/co-fair sets for atomic $\overline{\mathsf{SFS}}$ processes 134
4.23 A symbolic representation of the adjacency matrix and the path matrix of an atomic $\overline{\mathsf{SFS}}$ process 135
4.24 SAT encoding of $\Phi(P)$ 137
4.25 SAT encoding of $\delta(P)$ 137
4.26 BDD encoding of $\Phi(P)$ 138
4.27 BDD encoding of $\delta(P)$ 138
4.28 BDD encoding of $\mathsf{N}_X(P)$ 139
4.29 Infinite counters 139

5.1 CEGAR framework 152
5.2 $N(b,\mathcal{N})$ for b bad initial a_n after $\langle a_0, \ldots, a_{n-1}\rangle$ 165
5.3 $N(b,\mathcal{N})$ for b bad minimal acceptance B after $\langle a_0, \ldots, a_{n-1}\rangle$ 166
5.4 $N(b,\mathcal{N})$ for b bad divergence after $\langle a_0, \ldots, a_{n-1}\rangle$ 167
5.5 $B(b,\mathcal{N})$ for b bad lasso trace $\langle a_0, \ldots, a_{r-1}\rangle^\frown\langle a_r, \ldots, a_n\rangle^\omega$ 168
5.6 A spurious lasso trace 177
5.7 CEGAR loop for Milner's scheduler: part I 188
5.8 CEGAR loop for Milner's scheduler: part II 189

List of Tables

2.1 Visible and invisible actions 24
2.2 Denotational models 29

3.1 Bounded refinement checking: experimental results for peg solitaire 82
3.2 Bounded refinement checking: experimental results for the chess knight tour 83
3.3 Bounded refinement checking: experimental results for Hamiltonian cycle detection 83
3.4 Bounded refinement checking: experimental results for the lights-off puzzle 84
3.5 Bounded refinement checking: experimental results for the Hanoi towers . . 84
3.6 Bounded refinement checking: experimental results for violated security protocols 85
3.7 Temporal k-induction: experimental results 86

4.1 Acceptance conditions for ω-automata 132
4.2 Static livelock analysis: experimental results for infinite-state processes . . . 140
4.3 Static livelock analysis: experimental results for $\overline{\mathsf{SFS}}$ processes 141

5.1 CEGAR framework: experimental results 181
5.2 CEGAR framework: experimental results for the dining philosophers. . . . 184
5.3 CEGAR framework: comparison of different strategies for trace refinement. 185

List of Algorithms

1 Refinement checking in FDR 53
2 Leaf compression 56
3 Inductive compression 57
4 Temporal k-induction 66
5 Static analysis: computing fair/co-fair sets of events 134
6 CEGAR framework: obtaining all spurious executions of finite behaviours . 169
7 CEGAR framework: obtaining all spurious executions of lasso traces 169
8 CEGAR framework: simulating abstract executions 170

Chapter 1

Introduction

Model checking [CE81, QS82, CGP99, BK08] is a powerful automatic formal verification technique for establishing correctness of systems. It has been widely used in industry, e.g., NASA, Microsoft, Intel, Cadence, Bell Labs, etc., for analysing and verifying complex safety-critical systems in application domains such as avionics, medicine and computer security, where manual testing is infeasible and even minor errors could have dire consequences.

The foundations of model checking were established by Clarke and Emerson [CE81] in 1981 and independently by Queille and Sifakis [QS82] in 1982. For the significance of their work and contributions, in 2007 Clarke, Emerson and Sifakis won the prestigious ACM Turing Award.

Model checking relies on rigorous mathematical formalisms and algorithms to formally reason about the correctness of a program. In its general case, it requires a finite-state model (or a finite abstraction) of a system, capturing its possible behaviours, and a specification property, capturing the notion of correctness. Typically, the language description of the program is automatically compiled into a formal model, i.e., a mathematical structure such as an automaton, a transition system, a Kripke structure, a Petri net, etc. The specification is traditionally defined as a formula in some kind of temporal logic, but it can also take the form of an assertion statement in the program or, alternatively, be specified as an abstract design in the formal language of the system. In the latter case, the verification problem is called refinement checking.

The model checker performs analysis based on exhaustive systematic exploration of the state space of the system's model to either establish or refute that the system meets its specification. In the latter case, the model checker provides a counterexample—a witness illustrating a violating behaviour of the program. Depending on the underlying formalisms, the counterexample can take the form of a trace, a tree, a schedule, etc., that can be used for reproducing and fixing the bug, which is extremely useful in practice. Moreover, model checking is a representative of the push-button technology in the sense that in the vast majority of cases it is fully automatic and requires no human supervision or expertise, as opposed to other formal verification approaches such as theorem proving, for instance.

Model checking is very well suited not only for detecting bugs but also for formally verifying the correctness of systems. For a finite-state model of a system, model checking is complete, and therefore reliable when pronouncing a system correct. In case of infinite-state systems, the completeness of the algorithm usually[1] depends on whether or not the finite abstraction of the system preserves the correctness property under consideration.

The main challenge in applying model checking in practice is that the state space of the model of the system grows exponentially with the number and size of the variables it employs and can easily become too large to handle efficiently. The so-called *state-space explosion problem* puts serious restrictions on the size of systems that can be tractably analysed.

To combat the state-space explosion problem, a significant number of techniques have been proposed. Methods for decreasing the size of the generated state space and enhancing the model checking algorithm include abstraction [CGL94] (e.g., data, predicate, counter abstraction) usually embedded in a CEGAR [CGJ$^+$00] loop, invariant generation [McM03, SSS00], symmetry reduction [ES93, CFJ93, God99], state-space compression and hashing, etc. Many model checkers settle for a sound but incomplete algorithm by employing techniques such as bounded model checking [BCCZ99] for detecting some but not necessarily all the bugs in the system. Alternatively, model checkers based on static analysis or abstract interpretation [CC77] generally adopt complete but unsound algorithms that might raise false alarms but guarantee that a bug is never missed. Regarding state-space representation, the major dichotomy is between explicit and symbolic [BCM$^+$92, BCCZ99] model checking. Explicit model checking is based on explicit enumeration and examination

[1]There are instances of correct and complete model-checking algorithms that terminate on infinite-state structures based on reasoning on-the-fly about, e.g., well-quasi orders. Examples include Petri nets, lossy counter machines, 1-clock timed alternating automata, various term-rewrite systems, etc.

of individual states. Symbolic model checking relies on abstract representation of sets of states, generally as Boolean formulas, and properties are validated using techniques such as BDD manipulation or SAT-solving.

In our increasingly parallelised world, concurrency has become pivotal and seamlessly woven within programming paradigms, however, extremely challenging when it comes to modelling and establishing correctness of intended behaviour. Tools for model checking concurrent systems face severe limitations due to scalability problems arising from the need to examine all possible interleavings (schedules) of executions of parallel components. Every subsequent parallel component adds a further exponential blow-up to the state space of the system, amplifying enormously the state-space explosion problem. Moreover, concurrency poses additional challenges to model checking, giving rise to phenomena such as nondeterminism, deadlock, livelock, etc.

In addition to the above-mentioned model checking techniques, various techniques employing compositional reasoning have proven beneficial for moderating the severity of the state-space explosion problem in the setting of concurrent systems. Partial-order reductions [CGP99, Pel98, God95] exploit the interleaving model of concurrent systems and the notion of independence of concurrent events. They build upon the observation that behaviours that differ only in the order of independent concurrent events can be indistinguishable by the specification and can therefore be considered equivalent. Hence, only a subset of all possible interleavings can be explored. Other compositional techniques, targeted specifically at the formal verification of concurrent systems, include assume-guarantee reasoning [GL91, HQR00, BPG08], hierarchical state-space compression [RGG$^+$95, Ros11b], τ-confluence reduction [WK07], etc. Further surveys on model checking techniques and methods for state-space reduction can be found in [DKW08] and [Pel08].

Two fundamental paradigms for modelling concurrency are grounded upon communication using shared memory and communication using synchronous or asynchronous message passing. Hybrid concurrency models are adopted by multi-core GPUs, for instance, which exploit both shared memory and barrier synchronisation primitives. If parallel components are distributed, however, there is no physical shared memory involved, so distributed systems rely exclusively on the message passing model of concurrency. If adopting synchronous message passing, parallel components communicate by handshaking on the execution of certain actions, while still being able to perform independently, i.e., interleave, on other actions. Consequently, a sender process may use a channel to transmit data to a receiver process only if the latter is ready to accept data through that channel. Asynchronous message passing relies on typed channels of certain capacity (> 0). Hence, messages are buffered in channels and no handshaking is necessary.

Process algebras are a family of mathematical formalisms for modelling and reasoning about concurrent (reactive, distributed) systems in terms of the interactions of their component counterparts and the observable behaviours that they can exhibit, where the atomic units of observation are actions, also called events. Traditionally, process algebras are based on the synchronous message passing model of concurrency, i.e., component processes communicate explicitly by handshaking on events, as opposed to by modifying values of global shared variables. The most prominent process algebras to date are CCS (Calculus of Communicating Systems) [Mil80, Mil89], CSP (Communicating Sequential Processes) [Hoa85, Ros98, Ros11b], ACP (Algebra of Communicating Processes) [BK84] and the π-calculus [Mil99]. In all of those, using a high-level language, processes are defined compositionally, starting with atomic process constructs and combining those using operators such as choice, parallel and sequential composition, hiding, etc. A distinguishing mark of process algebras is that processes (programs) can be given formal algebraic semantics and can be compared for equivalence through an inference system based on a set of equational algebraic laws (axioms) that the basic operators satisfy. The algebraic laws provide a mechanism for calculating behaviours of processes using equational reasoning, hence the name algebra or calculus. A brief history and a survey of results in the field of process algebras can be found in [Bae05] and [Ace03], respectively.

Developed in the late 1970's by Tony Hoare, the process algebra CSP [Hoa85, Ros98, Ros11b] allows for a concise description and precise analysis of event-based concurrency. CSP is grounded on synchronous message passing as a fundamental means of inter-component communication, as opposed to shared-variable languages. Nevertheless, other models of concurrency are readily definable on top of CSP, e.g., shared-variable programs can be modelled by defining shared variables as processes [Ros11b].

In terms of syntax and semantics, among other differences with existing formalisms for modelling concurrent systems, CSP supports the usage of recursion, broadcast (multi-way) synchronisation, as well as hiding and renaming of events, both of which are powerful mechanisms for abstraction. Using a high-level language, processes can be defined compositionally and compiled in a hierarchical structure. This allows for a way of describing reactive systems that is usually very concise and much more economical in state space than shared-variable languages.

In terms of semantics, the primary means of understanding the meaning of CSP processes has been through denotational semantic models, although CSP also provides congruent algebraic and operational semantics of processes [Ros98]. In contrast, other process algebras mostly rely on algebraic (ACP) or operational semantics (CCS and the π-calculus). Denotational models map syntactic descriptions of processes into ordered mathematical structures. CSP supports a hierarchy of several such denotational semantic models that capture different types of behaviours, recording more or less information about a process. All denotational models are *compositional* in the sense that denotational values of component processes can be computed in terms of the denotational values of their subcomponents. Values of recursive processes can be obtained using standard fixed-point theory in the style of Scott and Strachey [SS71].

Traditionally in model checking, correctness properties are specified in some kind of temporal logic such as LTL, CTL, CTL*. In process algebras, modal logics are employed, e.g., the Hennessy-Milner logic [HM80] for CCS or other variants of modal logics for the π-calculus [MPW93]. In CSP correctness properties are defined as abstract designs of the systems, i.e., as processes, and the verification problem is called refinement checking. A process I refines a process S in a semantic model $\mathcal{M}$, denoted $S \sqsubseteq_{\mathcal{M}} I$ and interpreted as I is a valid implementation of the specification S in $\mathcal{M}$, if all behaviours of I observable in $\mathcal{M}$ are also valid behaviours of S. The refinement relation on processes is defined as a partial order in the semantic domain $\mathcal{M}$ and has a lot of useful properties to be exploited, among others monotonicity and transitivity, which allow for formalising a compositional and stepwise approach in development and verification.

Over the years, CSP has been widely used for modelling, analysis and verification of reactive systems. Various extensions of core CSP also appear in literature and practice, e.g., timed CSP [RR86, Oua00, Sch00, Ros11b], shared-variable CSP [Ros11b, SLD08], CSP with mobility [Ros11b, Ros10, ST05, VST09], probabilistic CSP [Low93, GW05], etc.

FDR [Ros94, G^+05], standing for *Failures Divergence Refinement*, is acknowledged as the primary tool supporting CSP [Hoa85, Ros98, Ros11b]. Originally released in 1992, FDR has evolved into a stable robust platform for modelling and reasoning about CSP processes over a number of semantic models with different expressive power—the *traces model*, which captures partial correctness properties, the *stable-failures model*, which additionally handles nondeterminism and deadlock, and the *failures-divergences model*, which captures a wide variety of total correctness properties[2]. The core of FDR is refinement checking in each of the semantic models, which is carried out on the level of the operational representation of the CSP processes. This is justified by the fact that semantic models are congruent to the standard operational semantics which interprets processes as labelled transition systems. The congruence theorems are presented and proven in [Ros98]. Refinement checking in FDR has been implemented using explicit state enumeration supplemented by state hashing, hierarchical state-space compression techniques [RGG^+95] and the partial-order reduction function *chase*.

Throughout the years, FDR has been widely used in research, teaching and industry [AJS05, qin, ver, BC05] for analysing safety-critical systems, and is well known for its use in security analysis [Low96]. Centered around FDR, Casper [Low98] is a prominent tool for analysing and verifying the correctness of security protocols, underlying the discovery of an attack on the Needham–Schroeder public key protocol in 1995 [Low95] and the verification of correctness of a fixed version of it in 1996 [Low96]. In recent years FDR has been used a number of times as the back-end of a verification engine aimed at notations other than CSP. Examples include Casper (see above), the shared-variable analyser SVA (see Chapters 18 and 19 of [Ros11b]), tools for reasoning about Statemate statecharts [RW06] and UML activity diagrams [ASST10], as well as a number of proprietary industrial tools. Attempts were also made for compiling CSP to other notations, e.g., translating probabilistic CSP [GW05] to input for the probabilistic model checker PRISM [HKNP06, KNP11].

[2]Recently, FDR has been extended to also support a number of more expressive CSP models, namely the revivals and refusal testing models, together with their divergence-strict analogues [AGL^+12].

1.1 Contributions of the Thesis

The objective of the work presented in this thesis is to:

1. Adapt and develop novel verification techniques for concurrent systems in the setting of the process algebra CSP.
2. Implement and integrate those techniques in FDR for enhancing its performance and increasing the scale of systems that can be tractably analysed.
3. Empirically evaluate and gain insight about which classes of systems are tackled successfully using each of those techniques.

In particular, we focus on techniques based on symbolic model checking, static analysis, abstraction schemes and CEGAR, all of them exploiting the compositionality inherent in CSP and targeting to combat the state-space explosion problem.

In the remainder of this section, we summarise our directions of research individually one by one. We also state our contributions and give insight about how we have met the objectives described above.

1.1.1 SAT-Based Trace Refinement Checking.

We investigate symbolic model-checking techniques based on Boolean satisfiability (SAT), which we adapt for the traces model of CSP, sufficient for verifying safety properties.

Theoretical Contributions. We tailor state-of-the-art SAT-based model checking techniques in a two-fold manner.

1. First, we adapt the bounded model checking framework (BMC) [BCCZ99] to the context of CSP and FDR yielding *bounded refinement checking* [POR09, POR12]. In our setting, we exploit a SAT solver to decide bounded language inclusion as opposed to bounded reachability of error states, as in most existing model checkers. Due to the harder problem to decide, the original syntactic translation of BMC to SAT cannot be applied directly and we adopt a semantic translation algorithm based on watchdog transformations [RGM$^+$03]. As a further contribution, we propose a new Boolean encoding of CSP processes resting on FDR's

hybrid two-level approach for calculating the operational semantics using supercombinators.

2. Due to its limited scope, the bounded refinement framework is only suitable for detecting bugs. We make our SAT framework complete by implementing temporal k-induction [SSS00, ES03b] on top of it. The latter aims at establishing inductiveness of properties and is capable of both bug finding and establishing the correctness of systems.

Implementation. We have implemented both BMC and temporal k-induction in a tool called SymFDR which builds upon FDR to obtain an alternative symbolic refinement engine. The symbolic engine [POR09, POR12] adopts FDR's implicit operational representation based on supercombinators [Ros11b] but explores this using SAT rather than explicitly. For both BMC and k-induction, SymFDR offers configurable support for a SAT solver (MiniSAT 2.0 [ES03a, EB05], PicoSAT 846 [Bie08] or ZChaff [MMZ+01], all used in incremental mode), Boolean encoding (one-hot or binary), traversal mode (forward or backward), etc. Within the BMC framework, SymFDR also offers support for configuring a SAT-call frequency, which specifies how often a SAT check is run to look for errors, relative to the number of steps to unfold the transition relation.

To the best of our knowledge, our implementation of the k-induction algorithm is the first attempt of applying unbounded SAT-based refinement checking to CSP, as well as k-induction to concurrent software systems, in general.

Evaluation. The BMC engine sometimes substantially outperforms the original explicit state-space exploration method adopted by FDR, especially for complex tightly-coupled combinatorial problems, as reported in [POR09, POR12]. For k-induction, the completeness threshold blows up in all cases, due to concurrency, and, therefore, high performance depends on whether or not the property is k-inductive for some small value of k. Hence, the SAT engine generally scales better only when counterexamples exist.

1.1.2 Static Analysis for Establishing Livelock Freedom.

Theoretical Contributions. We propose two frameworks for establishing livelock freedom of CSP processes using static analysis. Both frameworks employ a collection of rules based on the inductive syntactic structure of terms to either soundly classify a process as livelock free or report an inconclusive result which might be a false positive. Hence we trade accuracy for speed.

The general framework can handle the widest variety of CSP processes, including infinite-state ones. The analysis is based on reasoning about fixed points in terms of metric spaces, as well as on keeping track and overapproximating the fair sets of events of a process. In the standard metric on processes, the hiding operator fails to be nonexpansive. We introduce a new family of metrics parametrised by sets of visible events under which all CSP operators other than recursion are at least nonexpansive in all their arguments, including hiding. We prove that our semantic model equipped with our new metric forms a complete ultrametric space, the set of livelock-free processes being a closed subset thereof. We propose a system of rules that inductively generate a sound but incomplete set of metrics that guarantee (witness) that recursive processes have unique fixed points, and furthermore, that those unique fixed points are livelock-free (if this is the case).

We identify a class of structurally finite-state CSP processes for which we devise a simpler and more precise algorithm that forms the basis of our second framework. We propose a system of compositional rules for inductively generating a livelock flag together with a collection of fair/co-fair pairs of sets of events, the combination of which can be viewed as an abstraction of the system preserving livelock freedom. For minimal closed sequential components we compute exact abstractions by examining their transition systems in isolation. We become conservative and start losing precision only in the compositional rules for handling compound CSP processes, thereby allowing more elaborate and finer data to be computed efficiently. Our fairness notion of a collection of fair/co-fair sets of events has not been studied in the verification literature. As a further contribution, we suggest an algorithm for computing it given a transition system of a process. We also propose methods for efficiently encoding this algorithm into a symbolic circuit of size polynomial in the syntactic description of the process, which serves as a basis of our symbolic representation.

We note that the idea about using symbolic circuits for encoding polynomial-time algorithms is not bound to our specific process-algebraic framework and can be generalised and applied to various other contexts. A compact symbolic circuit can encode the input-output relationship of a PTIME algorithm for all possible inputs of the algorithm all at once. A translation of such a circuit into a BDD or into an input for SAT can be plugged into any symbolic implementation.

Implementation. We have implemented both frameworks in a tool called SLAP, which is an acronym for Static Livelock Analyser of Processes. Computationally, the crux of our algorithms revolves around the generation and manipulation of sets. The algorithms fit very naturally into a symbolic paradigm; hence SLAP is fully symbolic. The choice of an underlying symbolic engine is configurable, with support for using a SAT engine (based on MiniSAT 2.0), a BDD engine (based on CUDD 2.4.2), or running a SAT and a BDD analyser in parallel and reporting the results of the first one to finish.

We have also integrated the framework for analysing structurally finite-state processes in FDR [AGL+12], where it now constitutes an alternative back-end for establishing livelock freedom.

Evaluation. SLAP outperforms FDR by multiple orders of magnitude, while exhibiting only a low level of inconclusive results on a large suite of benchmarks. In addition, SLAP can also handle infinite-state processes, which are beyond the current capabilities of FDR.

1.1.3 Abstraction Schemes and CEGAR Framework for CSP and FDR.

We adapt the counterexample-guided abstraction refinement framework [CGJ+00], also known as CEGAR, to the setting of CSP and FDR. In this framework, an initially coarse abstraction of the system is iteratively refined (i.e., made more precise) on the basis of spurious counterexamples until either a genuine counterexample is derived or the property is proven to hold.

Theoretical Contributions. We develop a series of abstraction/refinement schemes for the traces, stable-failures and failures-divergences models of CSP and embed them into a *fully automated* and *compositional* CEGAR framework. We exploit the compositionality of CSP for the stages of initial abstraction, counterexample validation and abstraction refinement, extending the framework proposed in [COYC03, CCO$^+$05] and facilitated by the fact that supercombinators act on a higher level to control how leaf processes interact. Since abstracting models based on traces and abstracting models based on stable failures are triggered by two opposite forces, we propose abstraction and abstraction-refinement strategies for balancing those forces.

To the best of our knowledge, our work constitutes the first application of CEGAR, in its automated form, to the setting of CSP.

Implementation. We have implemented our algorithms in a prototype tool developed on top of FDR. Generally, we adopt lazy refinement strategies that yield coarser abstractions even though it takes a greater number of iterations to converge. The CEGAR framework will be available in subsequent releases of FDR.

Evaluation. Preliminary experiments indicate a significant enhancement in terms of performance when verifying both safety and liveness properties, including checks for livelock and deadlock. Different test cases benefit from different abstraction/refinement strategies, so we speculate that randomising the choice of those would produce consistently more favourable results.

1.2 Published Work

Some of the work in this thesis has been previously published in jointly authored papers. In this section, we map the contributions listed in Section 1.1 to the corresponding publications.

The SAT-based bounded trace refinement framework was first published in [POR09]. An extended version it, also reporting on the implementation and evaluation of the temporal k-induction algorithm, appeared later in [POR12].

Our work on static analysis for establishing livelock freedom was first presented at CONCUR 2011 [OPRW11], where it won the Best Paper Award. Subsequently, it got invited and accepted for publication at [OPRW13]. The extended journal version of the paper includes all the proofs of correctness and presents more in detail the framework for analysing structurally finite-state processes.

A tool paper reporting on various new substantial extensions of FDR appeared at CAV 2012 [AGL$^+$12]. Among other techniques, the paper summarises and briefly evaluates all three frameworks presented in this thesis: SAT-based refinement, static analysis for establishing livelock freedom and CEGAR. The paper also introduces FDR's ability to handle discrete and real-time processes, a few new semantic models of CSP, a new priority operator and a new compression technique based on divergence-respecting weak bisimulation.

1.3 Layout of the Thesis

The remainder of the thesis is organised as follows.

In Chapter 2, we introduce the relevant background material on CSP and FDR, as well as a number of mathematical formalisms and notations that we use throughout the dissertation.

Chapters 3, 4 and 5 contain the main contributions of the thesis.

In Chapter 3, we present a SAT-based framework for carrying out trace refinement, which includes both bounded refinement checking and temporal k-induction. We propose a strategy based on watchdog transformations for reducing the problem of bounded language containment, underlying CSP refinement checking, to the problem of bounded reachability, traditional for bounded model checking. We also present a new Boolean encoding of CSP processes mimicking FDR's hybrid operational semantics based on supercombinators.

Chapter 4 introduces a novel static analysis framework for establishing livelock freedom of CSP processes. In Section 4.5, we define a new family of metrics on CSP processes, parametric on sets of visible events, and state a number of theoretical results about them. In Section 4.6, we present our general framework, which is capable of handling the widest variety of CSP process. We provide a system of rules based on the syntactic structure of CSP terms that inductively generate a sound approximation of the sets of metrics that witness the existence of unique and livelock-free fixed points. In Section 4.7, we define a class of structurally finite-state processes, for which we introduce an alternative collection of compositional rules that soundly generate a livelock flag together with fair/co-fair characterisation of the infinite traces of a process. Given a transition system of a process, we suggest an algorithm for computing an exact fair/co-fair characterisation of the infinite traces of a process and we introduce an efficient encoding of the algorithm into a compact symbolic circuit.

In Chapter 5, we introduce a fully automatic and compositional CEGAR framework for CSP refinement checking. We present a series of abstraction/refinement schemes for the traces, stable-failures and failures-divergences model of CSP. We describe the main stages of the CEGAR loop and discuss techniques for carrying out them component-wisely.

In Chapter 6, we summarise and evaluate our work and propose a number of avenues for future research.

Finally, Appendix A provides proofs of correctness of our static analysis framework for establishing livelock freedom.

Chapter 2

Background Material

In this chapter we introduce the necessary background material for this thesis.

2.1 Formalisms and Notations

2.1.1 Alphabets, Words and Languages

A *finite alphabet* A is a finite non-empty set of symbols.

A *finite word* of length n over A is a sequence $u = \langle a_1, \ldots, a_n \rangle$ where $n \in \mathbb{N}$ and $a_i \in A$ for $1 \leq i \leq n$. In the special case when $n = 0$, $\varepsilon = \langle\rangle$ is called *the empty word.* An *infinite word* w over A is an infinite sequence of elements of A, i.e., a function $w : \mathbb{N} \to A$. We denote the set of all words of length n by A^n. We write $A^* = \bigcup_{n \in \mathbb{N}} A^n$ and A^ω to denote the set of all finite and infinite words, respectively. A *word* is a finite or an infinite word and $A^\infty = A^* \cup A^\omega$ denotes the set of all words over A. Given a finite word u and a finite or infinite word v, we denote by $u \frown v$ the concatenation of u and v. We write $u < v$ if u is a prefix of v. If u is a word and $C \subseteq A$, the projection of u on C is denoted by $u \restriction C$ and defined by:

$$\begin{array}{rcll} \langle\rangle \restriction C & = & \langle\rangle & \\ \langle c\rangle \frown u \restriction C & = & c \frown (u \restriction C) & \text{if } c \in C \\ \langle a\rangle \frown u \restriction C & = & u \restriction C & \text{if } a \notin C. \end{array}$$

A *language* L over an alphabet A is a set of words over A. A language L is *prefix-closed* if for every word $w \in L$ and every $v \in A^*$, $v < w$ implies that $v \in L$.

2.1.2 Labelled Transition Systems

Transition systems, along with automata and Kripke structures, are a fundamental formalism for modelling software and hardware systems.

A *labelled transition system (LTS)* is a quadruple $M = \langle S, \mathsf{init}, A, T\rangle$, where S is a finite set of states, $\mathsf{init} \in S$ is an initial state, A is a finite alphabet of labels (actions, events) and $T \subseteq S \times A \times S$ is a transition relation.

Throughout this section, we assume a fixed LTS $M = \langle S, s_0, A, T\rangle$, unless specified otherwise. For convenience, we often write $s \xrightarrow{a} s'$ to denote $(s, a, s') \in T$. Furthermore, we write $s \xrightarrow{a}$ if there exists $s' \in S$ such that $s \xrightarrow{a} s'$. We will write $\longrightarrow^*$ and $\longrightarrow^+$ to denote, respectively, the reflexive transitive and the transitive closure of the transition relation.

Given a word $w = \langle a_1 \dots a_n \dots\rangle$, an *execution fragment* $\rho = \langle s_0, a_1, s_1, a_2 \dots a_n, s_n \dots\rangle$ of M is a finite or infinite alternating sequence of states and labels, such that for all $i \geq 0$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. For a finite word w, we require that the sequence ends with a state. We refer to the sequence of states $\pi = \langle s_0 \dots s_n \dots\rangle$ as the *path* corresponding to ρ. A finite path $\pi = \langle s_0 \dots s_n\rangle$ is *simple* if all states along π are pairwise different, i.e., if for all $0 \leq i, j \leq n$, $i \neq j$ implies $s_i \neq s_j$. An execution fragment is *initial* if $s_0 = \mathsf{init}$.

An *execution* of M is any initial execution fragment.

A state $s \in S$ is *reachable* if there exists a finite execution of M ending in s. For any $n \in \mathbb{N}$, word $w = \langle a_1 \dots a_n\rangle \in A^*$ and states $s, s' \in S$, we say that s' is reachable from s through w and write $s \xrightarrow{w}^* s'$, if there exists a sequence of states $\langle s_0 \dots s_n\rangle$ such that $s_0 = s$, $s_n = s'$ and for all $0 \leq i < n$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. We denote by $\mathsf{Reach}(M, s, w) = \{s' \mid s \xrightarrow{w}^* s'\}$ the set of states in M reachable from s through w. We say that s' is reachable from s in M if there exists a word $w \in A^*$ such that $s' \in \mathsf{Reach}(M, s, w)$. A set of states $S' \subseteq S$ is *strongly connected* if any two states $s_i, s_j \in S'$ are mutually reachable from each other. A set of states $S' \subseteq S$ is a *strongly connected component (SCC)* if S' is strongly connected and no proper superset of S' is strongly connected.

Throughout the dissertation we also use the following notation. For any $s \in S$ and $a \in A$, we denote by $\mathsf{Post}(s, a) = \{s' \in S \mid s \xrightarrow{a} s'\}$ the set of all direct a-successors of s and by $\mathsf{Pre}(s, a) = \{s' \in S \mid s' \xrightarrow{a} s\}$ the set of all its direct a-predecessors. We use the standard lifting of Post and Pre to sets of states: $\mathsf{Post}(S, a) = \bigcup_{s \in S} \mathsf{Post}(s, a)$ and $\mathsf{Pre}(S, a) = \bigcup_{s \in S} \mathsf{Pre}(s, a)$.

The LTS M is *deterministic* if for all $s \in S$ and $a \in A$ there is at most one outgoing a-transition from s, i.e., if $| \mathsf{Post}(s, a) | \leq 1$. M is *total* if for all $s \in S$ and $a \in A$ there is at least one outgoing a-transition from s, i.e., if $| \mathsf{Post}(s, a) | \geq 1$. M is *finitely-branching* if for all $s \in S$ and $a \in A$ the set $\mathsf{Post}(s, a)$ is finite. Under the assumption that the alphabet A and the set of states S are finite, the LTS M is always finitely-branching.

2.1.3 Relations

For any $n \in \mathbb{N}$ with $n \geq 1$, an *n-ary relation* on nonempty sets $X_1, \ldots, X_n$ is a set of n-tuples $R \subseteq X_1 \times \ldots \times X_n$, where the *Cartesian product* $X_1 \times \ldots \times X_n$ is defined as follows:

$$X_1 \times \ldots \times X_n = \{\langle x_1, \ldots, x_n \rangle \mid x_i \in X_i \text{ for } i = 1, \ldots, n\}.$$

If $X_1 = \ldots = X_n = X$, then R is an n-ary relation on X.

For the rest of the section let us fix a non-empty set X and a binary relation $R \subseteq X \times X$ on X. For any $x, y \in X$, we write $x\ R\ y$ to denote $(x, y) \in R$. We say that R is:

reflexive if for all $x \in X$, $x\ R\ x$,

irreflexive if for all $x \in X$, $\neg(x\ R\ x)$,

symmetric if for all $x, y \in X$, $x\ R\ y$ implies $y\ R\ x$,

antisymmetric if for all $x, y \in X$, $x\ R\ y$ and $y\ R\ x$ implies $x = y$,

transitive if for all $x, y, z \in X$, $x\ R\ y$ and $y\ R\ z$ implies $x\ R\ z$.

For any $A, B \subseteq X$ we denote by $R(A)$ the set $\{b \in X \mid \exists\, a \in A \centerdot a\ R\ b\}$ and by $R^{-1}(B)$ the set $\{a \in X \mid \exists\, b \in B \centerdot a\ R\ b\}$. We use the standard lifting of R to words. If $a = \langle a_1, \ldots, a_k \rangle$ and $b = \langle b_1, \ldots, b_m \rangle$ are finite words over X, we write $a\ R\ b$ if $k = m$ and for all $1 \leq i \leq k$, $a_i\ R\ b_i$. If $a = \langle a_1, \ldots, a_k, \ldots \rangle$ and $b = \langle b_1, \ldots, b_k, \ldots \rangle$ are infinite words over X, we write $a\ R\ b$ if for all $i \in \mathbb{N}$, $a_i\ R\ b_i$.

2.1.3.1 Partial Orders and Lattices

In this section we recall standard definitions and facts concerning partial orders and lattices. We follow the accessible texts in [Cou05] and [Ros98].

Partial Orders. Let R be a binary relation on a non-empty set of elements X. We say that R is a *partial order* on X if R is reflexive, antisymmetric and transitive. R is a *strict partial order* on X if R is irreflexive and transitive. We usually denote partial orders by $\leq$, $\sqsubseteq$ or $\preceq$, and strict partial orders by $<$, $\sqsubset$ or $\prec$. If $\leq$ is a partial order on X, then the inverse relation $\geq$ defined as $x \geq y \mathrel{\widehat{=}} y \leq x$ is also a partial order on X, and similarly for strict partial orders.

Partial orders and strict partial orders are in one-to-one correspondence. If $\leq$ is a partial order on X, then the relation $<$ defined as

$$x < y \mathrel{\widehat{=}} y \leq x \text{ and } x \neq y,$$

is a strict partial order on X. Conversely, if $<$ is a strict partial order on X, then the relation $\leq$ defined as

$$x \leq y \mathrel{\widehat{=}} y < x \text{ or } x = y,$$

is a partial order on X. Therefore we will freely switch between the two formalisms.

Posets. A *partially ordered set* or a *poset* is a tuple $\langle X, \leq \rangle$ where $\leq$ is a partial order on X. The dual of a poset $\langle X, \leq \rangle$ is the poset $\langle X, \geq \rangle$, where $\geq$ is the inverse relation of $\leq$.

Let us fix a poset $\langle X, \leq \rangle$. Two elements $x, y \in X$ are *comparable* if $x \leq y$ or $y \leq x$. A subset $C \subseteq X$ is a *chain* of $\langle X, \leq \rangle$ if every two elements in C are comparable, i.e.,

$$\forall\, x, y \in C \centerdot x \leq y \text{ or } y \leq x.$$

A subset $A \subseteq X$ is an *antichain* of $\langle X, \leq \rangle$ if every two elements in A are incomparable, i.e.,

$$\forall\, x, y \in A \centerdot x \leq y \text{ implies } x = y.$$

An element $m \in X$ is a *minimal* element of $\langle X, \leq \rangle$ if there is no $x \in X$ with $x < m$. Similarly, $M \in X$ is a *maximal* element of $\langle X, \leq \rangle$ if there is no $x \in X$ with $M < x$.

In any poset $\langle X, \leq \rangle$, minimum and maximum elements may or may not exist. An element $\bot \in X$ is the *minimum* or *bottom* element of $\langle X, \leq \rangle$ if for all $x \in X$, $\bot \leq x$. Dually, $\top \in X$ is the *maximum* or *top* element of $\langle X, \leq \rangle$ if for all $x \in X$, $x \leq \top$. If they exist, bottom and top elements are unique because the relation $\leq$ is antisymmetric. If they do not exist, the poset may contain multiple minimal or maximal elements.

Let $\langle X, \leq \rangle$ be a poset and $S \subseteq X$. We say that an element $a \in X$ is:

- an *upper bound* for S if for every $x \in S$, $x \leq a$,
- a *lower bound* for S if for every $x \in S$, $a \leq x$,
- a *least upper bound* for S if it is an upper bound for S and, for any upper bound b of S, $a \leq b$,
- a *greatest lower bound* for S if it is a lower bound for S and, for any lower bound b of S, $b \leq a$.

The least upper bound and the greatest lower bound of S, if they exist, are denoted by $\bigsqcup S$ and $\bigsqcap S$, respectively. If they exist, $\bigsqcup S$ and $\bigsqcap S$ are unique.

Example 2.1.1. *Let us take an arbitrary non-empty set S and consider the powerset $\mathcal{P}(S)$ of S equipped with the subset relation $\subseteq$. $\langle \mathcal{P}(S), \subseteq \rangle$ is a poset with bottom element $\bot = \emptyset$ and top element $\top = S$. If $X \subseteq \mathcal{P}(S)$, then both $\bigsqcup X$ and $\bigsqcap X$ exist and are equal to $\bigcup X$ and $\bigcap X$, respectively.*

Semilattices and Lattices. A *join semilattice* is a triple $\langle X, \leq, \sqcup \rangle$, where $\langle X, \leq \rangle$ is a poset and every two elements $x, y \in X$ have a least upper bound $x \sqcup y$ in X. A *meet semilattice* is a triple $\langle X, \leq, \sqcap \rangle$, where $\langle X, \leq \rangle$ is a poset and every two elements $x, y \in X$ have a greatest lower bound $x \sqcap y$ in X. The elements $x \sqcup y$ and $x \sqcap y$ are called the *meet* and *join* of x and y, respectively. A *lattice* is quadruple $\langle X, \leq, \sqcup, \sqcap \rangle$, where $\langle X, \leq, \sqcup \rangle$ is a join semilattice and $\langle X, \leq, \sqcap \rangle$ is a meet semilattice. The poset $\langle \mathcal{P}(S), \subseteq \rangle$ from Example 2.1.1 is a lattice and is called the *powerset lattice* of S. In general, if $\langle X, \leq, \sqcup, \sqcap \rangle$ is a lattice and $Y \subseteq X$ is finite, then $\bigsqcup Y$ and $\bigsqcap Y$ exist. This is not necessarily the case if Y is infinite, however.

CPOs and Complete Lattices. A partial order $\langle X, \leq \rangle$ is a *complete partial order* (CPO) if it has a bottom element $\bot$ and every ascending chain $x_0 \leq x_1 \leq x_2 \ldots$ in X has a least upper bound $\bigsqcup_i x_i$ in X. Similarly, a lattice $\langle X, \leq, \sqcup, \sqcap \rangle$ is a *complete lattice* if *any* subset Y of X, finite or infinite, has a least upper bound $\bigsqcup Y$ in X, which is true precisely whenever *any* subset Y of X has a greatest lower bound $\bigsqcap Y$ in X. A complete lattice is always non-empty and has both a top and a bottom element. The powerset lattice from Example 2.1.1 is a complete lattice. Every finite lattice is a complete lattice.

Functions and Fixed Points. Let $f : X \mapsto X$ be a function on a non-empty set X. An element $a \in X$ is a *fixed point* of f if $f(a) = a$. An arbitrary function f can have $0, 1$ or multiple fixed points.

Let now $f : X \mapsto X$ be a function on X and $\langle X, \leq \rangle$ be a partial order. We say that f is *monotone* if for all $x, y \in X$, $x \leq y$ implies $f(x) \leq f(y)$. A fixed point $a \in X$ is the *least fixed point* of f if a is a fixed point of f, and for any fixed point b of f, $a \leq b$. Dually, a fixed point $a \in X$ is the *greatest fixed point* of f if a is a fixed point of f, and for any fixed point b of f, $b \leq a$. We denote the least and the greatest fixed point of f, if they exist, by μf and νf, respectively.

Theorem 2.1.2 (Knaster-Tarski fixed-point theorem)**.** *Let $\langle X, \leq \rangle$ be a complete lattice and $f : X \mapsto X$ be a monotone function on X. Then f has a least fixed point given by $\bigsqcap\{x \mid f(x) \leq x\}$ and a greatest fixed point given by $\bigsqcup\{x \mid x \leq f(x)\}$.* $\square$

Let now $f : X \mapsto X$ be a function on X and $\langle X, \leq \rangle$ be a CPO. We say that f is *continuous* if, whenever $x_0 \leq x_1 \leq x_2 \dots$ is an ascending chain of elements from X, $f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i)$. If f is continuous, then f is monotonic although the reverse is not necessarily true.

Theorem 2.1.3 (Kleene's fixed-point theorem)**.** *Let $\langle X, \leq \rangle$ be a CPO and $f : X \mapsto X$ be a continuous function on X. Then f has a least fixed point given by $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$.* $\square$

2.1.3.2 Equivalence Relations and Partitions

Equivalence Relations. A binary relation R on X is an *equivalence relation* on X if R is reflexive, symmetric and transitive. In case of equivalence relations, we often use the symbols $\sim$ or $\equiv$ instead of R.

For an arbitrary $x \in X$, the *equivalence class* of x under R is denoted by $[x]^R$ and is defined as $[x]^R = \{x' \in X \mid x \; R \; x'\}$. Then for any $x, y, z \in X$, $x \; R \; y$ if and only if $[x]^R = [y]^R$, which in turn holds if and only if $[x]^R \cap [y]^R \neq \emptyset$.

We write X/R to denote the *quotient* of X with respect to R, i.e., the set $\{[x]^R \mid x \in X\}$ of all equivalence classes induced in X by R.

Given equivalence relations R and R' on X, we say that R' is a *refinement* of R if for every $x \in X$, $[x]^{R'} \subseteq [x]^R$. In this case we also say that R' is *finer* than R or that R is *coarser* than R'. R' is a *proper refinement* of R if R' is a refinement of R and $R' \neq R$.

Partitions. Given a non-empty set X, a *partition* Π of X is a set of non-empty sets $\{X_i \mid i \in I\}$ such that $\bigcup_{i \in I} X_i = X$ and for every $i, j \in I$, if $i \neq j$, then $X_i \cap X_j = \emptyset$. In other words, P is a set of non-empty pairwise-disjoint subsets of X that cover the whole of X. The elements of Π are called *blocks* of the partition.

Similarly to equivalence relations, given two partitions Π and Π' of X, we say that Π' is a *refinement* of Π if every block B' of Π' is a subset of a block B of Π. If in addition $\Pi \neq \Pi'$, then Π' is a *proper refinement* of Π. In this case we say that Π' is *finer* than Π or that Π is *coarser* than Π'.

Correspondence Between Equivalence Relations and Partitions. In fact, equivalence relations and partitions are in one-to-one correspondence. If R is an equivalence relation on X, then the quotient X/R is a partition of X. Conversely, if $\Pi = \{X_i \mid i \in I\}$ is a partition of X, then the relation R defined as $\{(x, y) \mid \exists\, i \in I \,\textbf{.}\, x, y \in X_i\}$ is a well-defined equivalence relation on X.

2.1.4 Propositional Logic

Let X be a countably-infinite set of Boolean variables and $x \in X$. The set of well-formed *Boolean formulas* is defined inductively by the following grammar:

$$\varphi ::= \mathrm{true} \;\mid\; \mathrm{false} \;\mid\; x \;\mid\; \varphi_1 \wedge \varphi_2 \;\mid\; \neg\varphi \ .$$

Other Boolean operators such as OR ($\vee$), IMPLIES ($\rightarrow$) and EQUIVALENT TO ($\leftrightarrow$) are standardly derived using AND ($\wedge$) and NOT ($\neg$). A formula of the form x or $\neg x$ is called a *literal.* $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ are the *conjunction* and the *disjunction* of φ_1 and φ_2, respectively.

An *assignment* or a *valuation* is a function ν mapping the variables in X to the set of truth values $\{\mathrm{true}, \mathrm{false}\}$. Given a Boolean formula φ over a set of variables from X, we use the standard semantics of propositional formulas to determine whether φ evaluates to true under ν, which we denote by $\nu \models \varphi$.

$$\begin{array}{lcl}
\nu \models \text{true} & & \\
\nu \not\models \text{false} & & \\
\nu \models x & \Leftrightarrow & \nu(x) = \text{true} \\
\nu \models \neg\varphi & \Leftrightarrow & \nu \not\models \varphi \\
\nu \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & \nu \models \varphi_1 \text{ and } \nu \models \varphi_2
\end{array}$$

A formula φ is *satisfiable* if there exists an assignment of its variables under which it evaluates to true. If no such assignment exists, φ is *unsatisfiable* (or a *contradiction*). A formula φ is *valid* (or a *tautology*) if it evaluates to true under all possible assignments. As a trivial consequence, φ is valid if and only if $\neg\phi$ is a contradiction.

We say that two formulas φ_1 and φ_2 are *equivalent* and write $\varphi_1 \equiv \varphi_2$ if for any assignment ν, $\nu \models \varphi_1$ if and only if $\nu \models \varphi_2$. φ_1 and φ_2 are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

A formula is in *negation normal form* (NNF) if the only Boolean operators that it contains are AND, OR and NOT, and negation occurs only in front of Boolean variables. By repeatedly applying De Morgan's rules and the double-negation rule, every Boolean formula φ can be transformed into an equivalent formula in NNF of size linear in the size of φ.

A formula is in *conjunctive normal form* (CNF) if it is a conjunction $\bigwedge_i c_i$ of clauses, where each *clause* c_i is a disjunction $\bigvee_j l_{i,j}$ of literals. A clause containing just a single literal is called a *unary clause*. It is well-known that every Boolean formula φ can be transformed into an equivalent formula in CNF although potentially increasing φ's size exponentially. On the other hand, φ can be transformed to an equisatisfiable formula in CNF with only a linear blow-up in its size. One way of carrying out the transformation is via an algorithm known as *Tseitin encoding* [Tse68, BKWW08, KS08].

Tseitin encoding is an inductive algorithm for converting a Boolean formula φ into an equisatisfiable formula in CNF. The algorithm works top-down on the structure of φ by introducing a fresh Boolean variable x_i for each non-literal subformula φ_i of φ and generating clauses requiring that the variable be equivalent to the subformula. The original formula φ is satisfiable if and only if the set of clauses generated in the above manner, in conjunction with the unary clause containing the fresh variable introduced for φ, is satisfiable. The formulas are equisatisfiable but not equivalent because of the fresh variables introduced. For each subformula φ_i of φ the clauses used as constraints are generated with respect to the (topmost) Boolean connective of φ_i. The rules are illustrated in Figure 2.1

[BKWW08], where x denotes the fresh variable introduced for φ_i, and y and z correspond to the variables that represent the subformulas of φ_i. They might be fresh variables as well, introduced inductively by the algorithm, or literals, if leaves.

$$\begin{array}{lcl}
x \leftrightarrow \neg y & \equiv & (\neg x \vee \neg y) \wedge (y \vee x) \\
x \leftrightarrow (y \vee z) & \equiv & (\neg y \vee x) \wedge (\neg z \vee x) \wedge (\neg x \vee y \vee z) \\
x \leftrightarrow (y \wedge z) & \equiv & (\neg x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z \vee x) \\
x \leftrightarrow (y \rightarrow z) & \equiv & (x \vee y) \wedge (\neg z \vee x) \wedge (\neg x \vee \neg y \vee z) \\
x \leftrightarrow (y \leftrightarrow z) & \equiv & (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg z \vee y) \wedge (\neg y \vee \neg z \vee x) \wedge (y \vee z \vee x)
\end{array}$$

Figure 2.1: Tseitin encoding rules

In order to reduce the size of the CNF formulas obtained using Tseitin's transformation algorithm, various Boolean optimisations can be applied. For instance, as suggested in [KS08], Tseitin encoding rules can transformed in a way that disjunctions and conjunctions of *multiple* subformulas be encoded by introducing a single fresh Boolean variable. The rules are presented in Figure 2.2.

$$\begin{array}{lcl}
x \leftrightarrow (y_1 \ \vee \ \ldots \ \vee \ y_n) & \equiv & (\neg x \vee y_1 \vee \ldots \vee y_n) \wedge (x \vee \neg y_1) \wedge \ldots (x \vee \neg y_n) \\
x \leftrightarrow (y_1 \ \wedge \ \ldots \ \wedge \ y_n) & \equiv & (x \vee \neg y_1 \vee \ldots \vee \neg y_n) \wedge (\neg x \vee y_1) \wedge \ldots (\neg x \vee y_n)
\end{array}$$

Figure 2.2: Tseitin encoding rules: multiple disjunctions and conjunctions

In addition, if the original formula is in NNF as defined above, the number and size of clauses of the equisatisfiable CNF formula can be further reduced by using a one-sided Tseitin encoding, also known as Plaisted-Greenbaum encoding [KS08, PG86]. The latter means that for $n \geq 1$, all constraints of the form $x \leftrightarrow \textsc{op}(y_1, \ldots, y_n)$ from Figures 2.1 and 2.2 can be replaced by $x \rightarrow \textsc{op}(y_1, \ldots, y_n)$, with the corresponding encoding as a set of clauses. This is justified by the fact that NNF formulas are monotonic with respect to satisfiability.

2.2 CSP

The general problem we investigate is *refinement checking* in process-algebraic settings and, more specifically, in the context of CSP [Hoa85, Ros98, Ros11b].

In process algebras, systems are modelled as interactions of a collection of processes, communicating with each other and with the outer world via synchronous message passing, as opposed to shared variables. Using a high-level language, processes are defined compositionally and compiled into a hierarchical structure, starting with atomic process constructs and combining those using operators such as choice, parallel and sequential composition, hiding, etc. This allows for a way of describing reactive systems that is usually very concise and much more economical in state space than shared-variable languages.

Developed in the late 1970's by Hoare, CSP is one of the three original process algebras. It allows for the precise description and analysis of event-based concurrency. An advantage of the CSP framework is that it offers a well-developed syntax, algebraic and operational semantics, a hierarchy of congruent denotational semantic models, as well as a formal theory of refinement and compositional verification. In terms of syntax and semantics, among other differences with existing formalisms for modelling concurrent systems, CSP supports the usage of broadcast communication, recursion, as well as hiding and renaming of events, both of which are powerful mechanisms for abstraction.

Unlike in conventional model checking, where specifications are generally defined as temporal-logic formulae, in CSP specifications are defined as abstract designs of the systems, i.e., as processes, which allows for a stepwise development process. The refinement checking procedure decides whether the behaviours of the system are a subset of the behaviours of the specification, i.e., whether the system *refines* the specification. Hence, the verification problem reduces to checking for reverse containment of behaviours and, therefore, to reverse language inclusion.

2.2.1 Syntax

In CSP, processes interact with each other and an external environment by communicating (instantaneous) events. Events can be atomic or can model exchange of data along typed channels. Compound events consist of a channel name and a finite sequence of data components from predefined types. We refer to both atomic and compound events as events. More than one process may have to cooperate in the performance of an event, i.e., handshake on it. It is standard to distinguish between visible events that might need the cooperation of other processes or the environment and invisible internal actions that occur silently, are not observable or controllable outside a process and model an internal computation such as resolving of nondeterminism, unfolding of a recursion, abstraction of details.

Let Σ be a finite alphabet of visible events with $\tau, \checkmark \notin \Sigma$. τ denotes the invisible silent action and $\checkmark$—a successful termination of a process—a special action which is visible but uncontrollable from outside and can only occur last. Table 2.1 characterises the different types of actions with respect to their visibility and their ability to exert control from outside a process.

Event	Visible to the environment	Controlled by the environment
$a \in \Sigma$	yes	yes
τ	no	no
$\checkmark$	yes	no

Table 2.1: Visible and invisible actions

We write $\Sigma^{\checkmark}$ to denote $\Sigma \cup \{\checkmark\}$ and $\Sigma^{*\checkmark}$ to denote the set of finite sequences of elements from Σ which may end with $\checkmark$. In the notation below, we have $a \in \Sigma$, $A \subseteq \Sigma$ and b is a Boolean expression. R denotes a binary (renaming) relation on Σ. Its lifting to $\Sigma^{\checkmark}$ is understood to relate $\checkmark$ to itself. For a given process P, we denote by $\alpha_P \subseteq \Sigma^{\checkmark}$ the set of all visible events that P can perform. The variable X is drawn from a fixed infinite set of process variables. The channel c is allowed to transfer data from a data type T and $v \in T$. We recall the *core* syntax of CSP. In general, CSP is a guarded command language [Dij75].

Definition 2.2.1. *CSP terms are constructed according to the following grammar:*

$$P ::= \mathit{STOP} \mid a \longrightarrow P \mid c?x \longrightarrow P(x) \mid c!v \longrightarrow P \mid \mathit{SKIP} \mid P_1 \sqcap P_2 \mid P_1 \mathbin{\Box} P_2 \mid$$
$$P_1 \underset{A}{\|} P_2 \mid P_1 \mathbin{⨟} P_2 \mid P \setminus A \mid P[\![R]\!] \mid \textsf{if } b \textsf{ then } P_1 \textsf{ else } P_2 \mid X \mid X = P(X) \mid \mathit{DIV} \ .$$

We give intuition about the core CSP operators below.

- STOP is the deadlocked process, i.e., a process that is not capable of communicating any visible or τ actions.
- The prefixed process $a \longrightarrow P$ initially offers to engage in the event a and subsequently behaves like P. To model compound events, $c?x \longrightarrow P(x)$ is allowed to input any value x of type T on the channel c and subsequently evolve to $P(x)$. Similarly, the

process $c!v \longrightarrow P$ outputs a certain value v through the channel c and then starts behaving like P. In general, by using channels, processes can simultaneously input and output multiple data values withing a single event.

- *SKIP* represents successful termination and is willing to communicate $\checkmark$ at any time.

- $P_1 \,\Box\, P_2$ denotes the external choice of P_1 and P_2, whereas $P_1 \sqcap P_2$ models its internal (or nondeterministic) alternative. In the former case the choice is resolved by the environment, while in the latter—nondeterministically.

- The parallel composition $P_1 \underset{A}{\parallel} P_2$ can communicate an event from A only if both P_1 and P_2 are willing to do so—it is required that P_1 and P_2 synchronise (i.e., handshake) on all events in A and behave independently of each other with respect to all other events. In practice, it is common to synchronise P_1 and P_2 on the set of their shared events, i.e., use $A = \alpha_{P_1} \cap \alpha_{P_2}$. In the special case when $A = \emptyset$, we refer to $P_1 \parallel\!| \, P_2 = P_1 \underset{\emptyset}{\parallel} P_2$ as the interleaving of P_1 and P_2. Alphabets of processes can also be given explicitly: the alphabetised parallel $P_1 \;{}_{\alpha_{P_1}}\!\parallel_{\alpha_{P_2}}\; P_2$ corresponds to $P_1 \underset{\alpha_{P_1} \cap \alpha_{P_2}}{\parallel} P_2$. The operator $\underset{A}{\parallel}$ is called generalised parallel because all other parallel alternatives are derivable from it.

- $P \mathbin{⨾} Q$ is the sequential composition of P and Q: it denotes a process which behaves like P until P chooses to terminate (silently), at which point the process seamlessly starts to behave like Q.

- $P \setminus A$ is a process which behaves like P but with all communications in the set A hidden, i.e., turned into internal τ actions. Hence, the A events in P become invisible and uncontrollable by other processes or the environment by means of synchronisation.

- The renamed process $P[\![R]\!]$ derives its behaviours from those of P in that, whenever P can perform an event a, $P[\![R]\!]$ can engage in any event b such that $a\ R\ b$.

- The conditional process $\mathsf{if}\ b\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ behaves like P_1 or P_2 depending on whether the Boolean expression b evaluates to true or false.

- $X = P(X)$ denotes a recursive process.

- Lastly, the process DIV represents livelock, i.e., a process caught in an infinite loop of silent events.

FDR supports the machine-readable language CSP$_M$ which extends core CSP with several further operators and an extensive functional language. In terms of data types, CSP$_M$ offers support for defining and manipulating Booleans, integers, tuples, sequences, sets, user-defined types, etc. Data types can be combined and nested arbitrarily as long as they remain finite. Both processes and events can be parametric on any finite number of those data types and, furthermore, processes can also be parametric on channel names. CSP$_M$ is augmented with support for using pattern matching, higher-order functions, local definitions, lambda terms, etc., which further allows for the concise modelling of complex hierarchical systems.

Regarding CSP operators, CSP$_M$ offers support for handling several derived operators such as timeout, interrupt, piping, enslavement, etc. It also provides means for defining replicated versions of all associative binary CSP operators. We list several examples below (where I denotes a finite indexed set and $A \subseteq \Sigma$):

- $?x : A \longrightarrow P(x)$: replicated prefixed process
- $||_{i \in I} \, \alpha_i \; @ \; P_i$: replicated alphabetised parallel
- $\sqcap_{i \in I} \; @ \; P_i$ or $\sqcap_{a \in A} \, a \longrightarrow P(a)$: replicated internal choice

2.2.2 Example: Milner's Scheduler

Given a number $N \in \mathbb{N}$, a scheduler must arrange two classes of events $a.i$ and $b.i$ for $i \in \{0, \ldots, N-1\}$, conforming to the following two requirements.

1. The $a.i$'s should occur in strict rotation, i.e., $\underbrace{a.0, a.1, \ldots, a.N-1}, \underbrace{a.0, a.1. \ldots, a.N-1}, \ldots$
2. There should be precisely one $b.i$ between each pair of $a.i$'s.

In CSP, Milner's scheduler can be modelled as a ring of cell processes synchronised using extra events $c.i$, as illustrated in Figure 2.3 for $N = 5$.

An abstracted CSP script for Milner's scheduler establishing the rotation specification is presented in Figure 2.4, where $i \oplus 1$ and $i \ominus 1$ denote, respectively, $(i+1)\%N$ and $(i-1)\%N$.

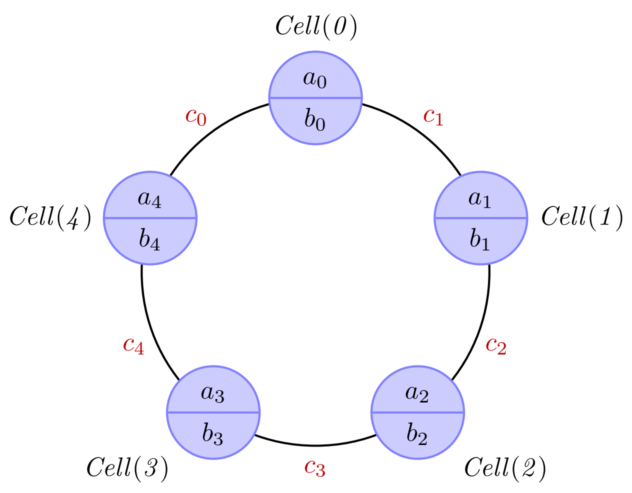


Figure 2.3: Milner's scheduler for $N = 5$

$$\begin{array}{ll}
Cell(0) = a.0 \longrightarrow c.1 \longrightarrow b.0 \longrightarrow c.0 \longrightarrow Cell(0) & \\
Cell(i) = c.i \longrightarrow a.i \longrightarrow c.i \oplus 1 \longrightarrow b.i \longrightarrow Cell(i) & \text{if } i > 0 \\
 & \\
Scheduler = (Cell(0) \parallel Cell(1) \parallel \ldots \parallel Cell(N-1)) \setminus \{\mid c \mid\} & \\
 & \\
ASpec(i) = a.i \longrightarrow ASpec(i \oplus 1) & \\
Spec = ASpec(0) & \\
 & \\
\text{assert } Spec \sqsubseteq_T Scheduler \setminus \{\mid b \mid\} &
\end{array}$$

Figure 2.4: CSP syntax: Milner's scheduler

In order to model Milner's scheduler, we extend the alphabet $\alpha_{Cell(i)} = \{a.i, b.i\}$ of every process $Cell(i)$ with extra events $c.i$ and $c.i \oplus 1$. The process $Cell(i)$ uses $c.i$ and $c.i \oplus 1$ to synchronise with its neighbouring processes $Cell(i \ominus 1)$ and $Cell(i \oplus 1)$, respectively. $Cell(0)$ is defined in a slightly different way as the a-sequence should start with $a.0$. The scheduler is constructed by composing all the cells in parallel and hiding all c-events on top as they have been introduced solely for synchronisation purposes. Within $Scheduler$, the operator $\parallel$ corresponds to synchronising the two argument processes on the set of their common events and is fully associative. Hence, a cell can perform an event only if all other cells that have the same event in their alphabet are also offering to do so. Using the replicated alphabetised parallel operator, the scheduler can also be modelled as follows:

$$\mathit{Scheduler} = \;\big|\big|_{i \in \{0,\ldots,4\}} \{a.i, b.i, c.i, c.i \oplus 1\} \mathbin{@} \mathit{Cell}(i).$$

$\{|\,c\,|\}$ is a shorthand for $\{c.i \mid i \in \mathbb{N}\}$.

We give a rough idea of how *Scheduler* works and why it preserves the rotation specification, modelled as *Spec*. The only process that can initially perform an event is *Cell*(*0*)—for all $i > 0$, *Cell*(i) is blocked as it needs *Cell*($i - 1$) to also offer $c.i$. After *Cell*(*0*) communicates $a.0$, the only thing that can happen next is *Cell*(*0*) and *Cell*(*1*) synchronising on $c.1$, thereby enabling *Cell*(*1*) to perform $a.1$. Concerning the sequence of a's, the same process is repeated around the ring as, synchronising on $c.i \oplus 1$, *Cell*(i) passes a token to *Cell*($i \oplus 1$) to signify that it is *Cell*($i \oplus 1$)'s turn to contribute an $a.i \oplus 1$. Obviously, the second requirement for the scheduler is captured as well, also in the most general way.

2.2.3 A Hierarchy of Denotational Models

Traditionally, the primary means of understanding CSP processes has been to use denotational (behavioural) models, whereby a process is identified with the set of observable behaviours that it can exhibit.

CSP supports a hierarchy of several such denotational semantic models. Different models describe different types of behaviours, providing more or less information about a process, with the natural trade-off between the amount of details recorded for a process and the complexity of working in the model. All denotational models are *compositional* in the sense that the denotational value (the set of possible behaviours) of each process can be computed in terms of the denotational values of its subcomponents. Values of recursive processes can be obtained using standard fixed-point theory in the style of Scott and Strachey (see Section 2.1.3.1).

The untimed models that we present in the thesis—the *traces model* $\mathcal{T}$ [Hoa80], which captures partial correctness properties, the *stable-failures model* $\mathcal{F}$ [BHR84], which additionally handles nondeterminism and deadlock, and the *failures-divergences model* $\mathcal{N}$ [Ace03], which captures a wide variety of total correctness properties—form a hierarchy with an increasing degree of expressiveness and detail. These models distinguish processes in terms of their sets of traces, failures and divergences, or combinations of these. Given a process

P and a semantic model $\mathcal{M}$, the denotational value $[\![P]\!]_{\mathcal{M}}$ of P in the model $\mathcal{M}$ is given in Table 2.2 below. The table also reflects the type of properties capturable in each of the models.

Model $\mathcal{M}$	$[\![P]\!]_{\mathcal{M}}$	Properties
$\mathcal{T}$	$\mathsf{traces}(P)$	safety
$\mathcal{F}$	$(\mathsf{traces}(P), \mathsf{failures}(P))$	liveness, deadlock freedom, nondeterminism
$\mathcal{N}$	$(\mathsf{failures}_\perp(P), \mathsf{divergences}(P))$	livelock freedom

Table 2.2: Denotational models

In the rest of the section we describe those three models in greater detail. We also introduce the notion of *refinement*. A process P refines a process Q in the model $\mathcal{M}$, denoted $Q \sqsubseteq_{\mathcal{M}} P$, if all behaviours of P observable in $\mathcal{M}$ are also valid behaviours of Q. In other words refinement between processes corresponds to reverse containment of behaviours. Consequently, if P refines Q, P can be plugged in every context where Q would work without compromising the correctness of the system. We can think of Q as a more abstract and nondeterministic design of a system (i.e., a specification) and we can view P as a more concrete and deterministic implementation of Q. From algebraic point of view, $Q \sqsubseteq_{\mathcal{M}} P$ if and only if $Q =_{\mathcal{M}} P \sqcap Q$.

Refinement is a partial order and has a lot of useful properties to be exploited, among others:

transitivity: if $P \sqsubseteq_{\mathcal{M}} Q$ and $Q \sqsubseteq_{\mathcal{M}} R$, then $P \sqsubseteq_{\mathcal{M}} R$.

monotonicity: if $C[.]$ is a process context, i.e., a CSP process definition with an empty slot for placing a process, then $P \sqsubseteq_{\mathcal{M}} Q$ implies $C[P] \sqsubseteq_{\mathcal{M}} C[Q]$.

Those two properties allow for *compositional* and *stepwise refinement*. Suppose C is a concurrent system with n components running in parallel. Then, if $\mathit{Spec} \sqsubseteq_{\mathcal{M}} C(S_1, \ldots, S_n)$ and for all $i \in \{1, \ldots, n\}$ $S_i \sqsubseteq_{\mathcal{M}} P_i$, then $\mathit{Spec} \sqsubseteq_{\mathcal{M}} C(P_1, \ldots, P_n)$

We consider two processes P and Q equivalent in a model $\mathcal{M}$ and write $P \equiv_{\mathcal{M}} Q$ if $P \sqsubseteq_M Q$ and $Q \sqsubseteq_{\mathcal{M}} P$. The relation $\equiv_{\mathcal{M}}$ forms an equivalence relation on the set of

CSP processes, which is, furthermore, a congruence with respect to all CSP operators other than recursion. The latter implies that for every CSP processes P, Q, P', Q' and every CSP operator $\oplus$ other than recursion, if $P \equiv_{\mathcal{M}} P'$ and $Q \equiv_{\mathcal{M}} Q'$, then $P \oplus Q \equiv_{\mathcal{M}} P' \oplus Q'$. Intuitively, $P \equiv_{\mathcal{M}} Q$ means that the processes P and Q are behaviorally indistinguishable in the model $\mathcal{M}$ and hence interchangeable.

Regarding values of recursions, all three semantic models form complete partial orders or complete lattices under the refinement order $\sqsubseteq$ or its inverse—the subset order $\subseteq$ of behaviours. In those structures, all CSP operators other than recursion are continuous in all their arguments. A recursive process is then interpreted as the least fixed point in the corresponding partial order (see Figure 2.5 and Section 2.1.3.1).

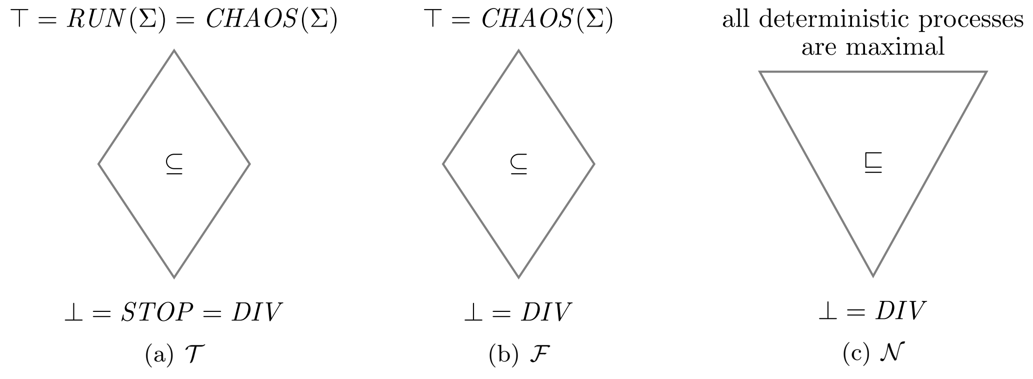

$\top = \mathit{RUN}(\Sigma) = \mathit{CHAOS}(\Sigma)$
$\subseteq$
$\bot = \mathit{STOP} = \mathit{DIV}$
(a) $\mathcal{T}$
$\top = \mathit{CHAOS}(\Sigma)$
$\subseteq$
$\bot = \mathit{DIV}$
(b) $\mathcal{F}$
all deterministic processes are maximal
$\sqsubseteq$
$\bot = \mathit{DIV}$
(c) $\mathcal{N}$

Figure 2.5: Order-theoretic structure of denotational models

Fixed points of recursive processes are actually proven to be unique provided that no hiding is used under recursion and every recursion is guarded by a visible event [Ros98]. This result is known as the *Unique Fixed Point rule (UFP)* and is obtained by reasoning about fixed points in terms of metric spaces.

2.2.3.1 The Traces Model.

In the simplest of all models, the traces model $\mathcal{T}$, a process P is identified with the set of its finite traces, denoted by $\mathsf{traces}(P) \subseteq \Sigma^{*\checkmark}$. Intuitively, a trace of a process is a sequence of visible actions that the process can perform. Naturally, the set of traces of a process is non-empty (it always contains the empty sequence) and prefix-closed. The traces model focuses only on the finite traces of a process, resting on the assumption that a process can perform an infinite trace if and only if it can perform all its finite prefixes.

We write $\mathcal{T}$ to denote all sets T satisfying the following axiom:

T1. $T \subseteq \Sigma^{*\checkmark}$ is non-empty and prefix-closed.

Given a process P, its denotation $[\![P]\!]_{\mathcal{T}} = \mathsf{traces}(P) \in \mathcal{T}$ is calculated by induction on the structure of P. We present the rules for the core CSP operators in Figure 2.6. The lifting of the renaming relation R to traces is carried out pointwise. The precise definition of $s \underset{A}{\parallel} t$ in the rule for parallel composition is presented in Figure 2.7 [Ros98]. The complete list of clauses can be found in [Ros98, Chap. 8] and [Ros11b].

$$\begin{aligned}
\mathsf{traces}(STOP) &= \{\langle\rangle\} \\
\mathsf{traces}(SKIP) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
\mathsf{traces}(DIV) &= \{\langle\rangle\} \\
\mathsf{traces}(a \longrightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle^\frown t \mid t \in \mathsf{traces}(P)\} \\
\mathsf{traces}(P \,\square\, Q) &= \mathsf{traces}(P) \cup \mathsf{traces}(Q) \\
\mathsf{traces}(P \sqcap Q) &= \mathsf{traces}(P) \cup \mathsf{traces}(Q) \\
\mathsf{traces}(P \,;\, Q) &= (\mathsf{traces}(P) \cap \Sigma^*) \cup \{t^\frown s \mid t^\frown\langle\checkmark\rangle \in \mathsf{traces}(P), s \in \mathsf{traces}(Q)\} \\
\mathsf{traces}(P \setminus A) &= \{t \upharpoonright (\Sigma \setminus A) \mid t \in \mathsf{traces}(P)\} \\
\mathsf{traces}(P[\![R]\!]) &= \{t \mid \exists\, s \in \mathsf{traces}(P) \bullet s\ R\ t\} \\
\mathsf{traces}(P \underset{A}{\parallel} Q) &= \bigcup\{s \underset{A}{\parallel} t \mid s \in \mathsf{traces}(P), t \in \mathsf{traces}(Q)\}
\end{aligned}$$

Figure 2.6: The model $\mathcal{T}$: inductive rules for calculating traces

Example 2.2.2. *Let us go back to the Milner's scheduler described in Figure 2.4. A trace of Cell(0) is any prefix of* $\langle a.0, c.1, b.0, c.0\rangle^*$.

In the traces model, a process *Impl* refines a process *Spec* if all sequences of events that *Impl* can communicate are also possible for *Spec*:

$$Spec \sqsubseteq_T Impl \quad \widehat{=} \quad \mathsf{traces}(Impl) \subseteq \mathsf{traces}(Spec).$$

The traces of a process specify what a process *may* do, but not what a process *must* do. For example, the traces model does not distinguish between the processes $P = a \longrightarrow P$ and $P' = a \longrightarrow P' \sqcap STOP$, neither does it distinguish between $Q = a \longrightarrow Q \,\square\, b \longrightarrow Q$ and $Q' = a \longrightarrow Q' \sqcap b \longrightarrow Q'$. However, both P and Q must offer infinitely many a's if

$$\begin{aligned}
s \underset{A}{\|} t &= t \underset{A}{\|} s \\
\langle\rangle \underset{A}{\|} \langle\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{A}{\|} \langle a\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{A}{\|} \langle b\rangle &= \{\langle b\rangle\} \\
\langle a\rangle^\frown s \underset{A}{\|} \langle b\rangle^\frown t &= \{\langle b\rangle^\frown u \mid u \in \langle a\rangle^\frown s \underset{A}{\|} t\} \\
\langle a\rangle^\frown s \underset{A}{\|} \langle a\rangle^\frown t &= \{\langle a\rangle^\frown u \mid u \in s \underset{A}{\|} t\} \\
\langle a\rangle^\frown s \underset{A}{\|} \langle a'\rangle^\frown t &= \{\} \quad \text{if } a \neq a' \\
\langle b\rangle^\frown s \underset{A}{\|} \langle b'\rangle^\frown t &= \{b^\frown u \mid u \in s \underset{A}{\|} \langle b'\rangle^\frown t\} \cup \{b'^\frown u \mid u \in \langle b\rangle^\frown s \underset{A}{\|} t\}
\end{aligned}$$

Figure 2.7: Interleaving operator on traces (where $s, t \in \Sigma^{*\checkmark}$, $A \subseteq \Sigma^{\checkmark}$, $a \in A$, $b \notin A$).

another process wanted to synchronise, whereas both P' and Q' might refuse an a at any point. Therefore, the traces model is sufficient for verifying all safety properties, but not liveness ones.

2.2.3.2 The Stable-Failures Model.

The stable-failures model $\mathcal{F}$ offers a finer distinction between processes by extending the traces model with the set of stable failures of a process, denoted by $\mathsf{failures}(P)$. In this model, a process is identified by the pair $(\mathsf{traces}(P), \mathsf{failures}(P))$.

A process is in a stable state if it cannot perform τ and $\checkmark$, which are out of the control of the environment. In a nutshell, $\mathsf{failures}(P)$ comprises all possible behaviours (t, X), where t is a finite trace leading to a stable state from which P can refuse to communicate any event from X. It is important to note that for a given trace t, there might be multiple stable failures (t, X_1), ..., (t, X_n), where the sets of events $X_1, \ldots, X_n$ are pairwise incomparable under set containment. For example, a nondeterministic process may accept t through multiple paths leading to different states from which different sets of events can be refused.

The stable-failures model $\mathcal{F}$ consists of all pairs (T, F) satisfying the following axioms (where $T \subseteq \Sigma^{*\checkmark}$ and $F \subseteq \Sigma^{*\checkmark} \times \mathcal{P}(\Sigma^{\checkmark})$):

T1. T is non-empty and prefix-closed.

T2. $(t, X) \in F$ implies $t \in T$.

F1. If $(t, X) \in F$ and $Y \subseteq X$, then $(t, Y) \in F$.

F2. If $(t, X) \in F$ and for all $a \in Y$, $t^\frown\langle a\rangle \notin T$, then $(t, X \cup Y) \in F$.

F3. $t^\frown\langle \checkmark\rangle \in T$ implies $\{\ (t^\frown\langle\checkmark\rangle, \Sigma^{\checkmark}), (t, \Sigma)\} \subseteq F$.

Given a process P, its denotation $[\![P]\!]_{\mathcal{F}} = (\mathsf{traces}(P), \mathsf{failures}(P)) \in \mathcal{F}$ is calculated by induction on the structure of P. We present the stable-failure rules for the core CSP operators in Figure 2.8 and the rules for deriving traces are the same as the ones for the traces model (see Figure 2.6). The complete list of clauses can be found in [Ros98, Chap. 8] and [Ros11b]. We note that the set $\mathsf{traces}(P)$ is recorded explicitly as part of $[\![P]\!]_{\mathcal{F}}$ because $\mathsf{failures}(P)$ only refers to traces that lead to stable states. For example, if we consider the process DIV, $\mathsf{failures}(DIV) = \emptyset$, but $\mathsf{traces}(DIV) = \{\langle\rangle\}$.

$$\begin{aligned}
\mathsf{failures}(STOP) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}\\
\mathsf{failures}(SKIP) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\}\\
\mathsf{failures}(DIV) &= \emptyset\\
\mathsf{failures}(a \longrightarrow P) &= \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle^\frown t, X) \mid (t, X) \in \mathsf{failures}(P)\}\\
\mathsf{failures}(P \mathrel{\Box} Q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \mathsf{failures}(P) \cap \mathsf{failures}(Q)\} \cup\\
&\quad \{(t, X) \mid t \neq \langle\rangle, (t, X) \in \mathsf{failures}(P) \cup \mathsf{failures}(Q)\} \cup\\
&\quad \{(\langle\rangle, X) \mid X \subseteq \Sigma, \langle\checkmark\rangle \in \mathsf{traces}(P) \cup \mathsf{traces}(Q)\}\\
\mathsf{failures}(P \sqcap Q) &= \mathsf{failures}(P) \cup \mathsf{failures}(Q)\\
\mathsf{failures}(P \mathbin{;} Q) &= \{(t, X) \mid t \in \Sigma^*, (t, X \cup \{\checkmark\}) \in \mathsf{failures}(P)\} \cup\\
&\quad \{(t^\frown s, X) \mid t^\frown\langle\checkmark\rangle \in \mathsf{traces}(P), (s, X) \in \mathsf{failures}(Q)\}\\
\mathsf{failures}(P \setminus A) &= \{(t \restriction (\Sigma \setminus A), X) \mid (t, A \cup X) \in \mathsf{failures}(P)\}\\
\mathsf{failures}(P[\![R]\!]) &= \{(t, X) \mid \exists\, s \in \mathsf{traces}(P) \bullet s\, R\, t, (s, R^{-1}(X)) \in \mathsf{failures}(P)\}\\
\mathsf{failures}(P \underset{A}{\parallel} Q) &= \{(u, X \cup Y) \mid \exists\, s\, \exists\, t \bullet u \in s \underset{A}{\parallel} t, (s, X) \in \mathsf{failures}(P), (t, Y) \in \mathsf{failures}(Q),\\
&\qquad X \restriction (A \cup \{\checkmark\}) = Y \restriction (A \cup \{\checkmark\})\}
\end{aligned}$$

Figure 2.8: The model $\mathcal{F}$: inductive rules for calculating stable failures

In the stable-failures model, a process *Impl* refines a process *Spec* if all traces and all failures that *Impl* can exhibit are acceptable for *Spec*:

$$Spec \sqsubseteq_F Impl \quad \widehat{=} \quad \mathsf{traces}(Impl) \subseteq \mathsf{traces}(Spec) \text{ and } \mathsf{failures}(Impl) \subseteq \mathsf{failures}(Spec).$$

The stable-failures model further allows for capturing liveness properties, such as deadlock freedom, nondeterminism, etc. A process P can deadlock after a trace t if (t, Σ) is a stable failure of P. The simplest example of a deadlocked process is the process $STOP$ and, in practice, deadlock is usually introduced by a collection of processes each of which being blocked and waiting for another blocked one to synchronise with. A process P defined over an alphabet of events Σ is deadlock-free if and only if the following refinement holds:

$$DF(\Sigma) \sqsubseteq_F P,$$

where $DF(\Sigma)$ is the most general and nondeterministic deadlock-free process over Σ:

$$DF(\Sigma) = \bigsqcap_{a \in \Sigma} a \longrightarrow DF(\Sigma).$$

A process P is nondeterministic if it can diverge, or if after some trace t it can both offer and refuse an event $a \in \Sigma$, i.e., if $t^\frown\langle a\rangle \in \mathsf{traces}(P)$ and $(t, \{a\}) \in \mathsf{failures}(P)$.

2.2.3.3 The Failures-Divergences Model.

The failures-divergences model $\mathcal{N}$ augments the stable-failures model by further distinguishing processes that can *diverge* (or *livelock*), i.e., engage in an endless unbroken internal activity. In this model, a process P is identified by the pair

$$(\mathsf{failures}_\perp(P), \mathsf{divergences}(P)),$$

where $\mathsf{divergences}(P)$ is the set of traces after which P can diverge and $\mathsf{failures}_\perp(P)$ is the set of stable failures, with an extension allowing a process to refuse anything after diverging:

$$\mathsf{failures}_\perp(P) = \mathsf{failures}(P) \cup \{(t, X) \mid t \in \mathsf{divergences}(P), X \subseteq \Sigma^{\checkmark}\}.$$

It is important to note that for any trace t that P can perform, either P reaches a stable state after t, or diverges after t or can terminate successfully after t (i.e., communicate $\checkmark$). Therefore all traces that P can perform, including those after which P can diverge, denoted by $\mathsf{traces}_\perp(P)$, can be derived from $\mathsf{failures}_\perp(P)$ in the following way:

$$\mathsf{traces}_\perp(P) = \{t \mid (t, \emptyset) \in \mathsf{failures}_\perp(P)\}.$$

The failures-divergences model $\mathcal{N}$ consists of all pairs (F, D) satisfying the following axioms (where $F \subseteq \Sigma^{*\checkmark} \times \mathcal{P}(\Sigma^{\checkmark})$ and $D \subseteq \Sigma^{*\checkmark}$ and T is derived from F as above):

F1. $T = \{t \mid (t, X) \in F\}$ is non-empty and prefix-closed.

F2. If $(t, X) \in F$ and $Y \subseteq X$, then $(t, Y) \in F$.

F3. If $(t, X) \in F$ and for all $a \in Y$, $t^\frown\langle a\rangle \notin T$, then $(t, X \cup Y) \in F$.

F4. If $t^\frown\langle\checkmark\rangle \in T$, then $(t, \Sigma) \in F$.

D1. If $t \in D \cap \Sigma^*$ and $s \in \Sigma^{*\checkmark}$, then $t^\frown s \in D$.

D2. $t \in D$ implies $(t, X) \in F$.

D3. $t^\frown\langle\checkmark\rangle \in D$ implies $t \in D$.

Since divergence is considered catastrophic, processes are considered to be behaving chaotically once they diverge. Hence, the set of divergences is defined to be postfix-closed and processes can refuse anything past a divergence point, as reflected in Axioms D1 and D2, respectively.

Given a process P, its denotation $[\![P]\!]_{\mathcal{N}} = (\mathsf{failures}_\perp(P), \mathsf{divergences}(P)) \in \mathcal{N}$ is calculated by induction on the structure of P. We present the divergence rules for the core CSP operators in Figure 2.9 and the rules for deriving the set $\mathsf{failures}_\perp(P)$ can be obtained from the rules for calculating stable failures in the stable-failures model (see Figure 2.8) by adding all possible refusals past divergent traces. The complete list of clauses can be found in [Ros98, Chap. 8] and [Ros11b]. In the last three rules in Figure 2.9, r ranges over $\Sigma^{*\checkmark}$, in accordance with Axiom D1.

In the failures-divergences model, the refinement relation $\sqsubseteq_{FD}$ is defined as reverse inclusion on both the failure and the divergence components:

$$\mathit{Spec} \sqsubseteq_{FD} \mathit{Impl} \mathrel{\widehat{=}} \mathsf{failures}_\perp(\mathit{Impl}) \subseteq \mathsf{failures}_\perp(\mathit{Spec}), \mathsf{divergences}(\mathit{Impl}) \subseteq \mathsf{divergences}(\mathit{Spec}).$$

A process P over an alphabet of events Σ is livelock-free if $\mathsf{divergences}(P) = \emptyset$, which is the case precisely whenever the following refinement relation holds:

$$\mathit{CHAOS}(\Sigma) \sqsubseteq_{FD} P,$$

where $\mathit{CHAOS}(\Sigma)$ is the most general and nondeterministic livelock-free process over Σ:

$$\mathit{CHAOS}(\Sigma) = (\textstyle\bigsqcap_{a \in \Sigma} a \longrightarrow \mathit{CHAOS}(\Sigma)) \sqcap \mathit{STOP}.$$

$$\begin{aligned}
\mathsf{divergences}(STOP) &= \emptyset\\
\mathsf{divergences}(SKIP) &= \emptyset\\
\mathsf{divergences}(DIV) &= \Sigma^{*\checkmark}\\
\mathsf{divergences}(a \longrightarrow P) &= \{\langle a\rangle^\frown t \mid t \in \mathsf{divergences}(P)\}\\
\mathsf{divergences}(P \mathbin{\Box} Q) &= \mathsf{divergences}(P) \cup \mathsf{divergences}(Q)\\
\mathsf{divergences}(P \sqcap Q) &= \mathsf{divergences}(P) \cup \mathsf{divergences}(Q)\\
\mathsf{divergences}(P \mathbin{;} Q) &= \mathsf{divergences}(P) \cup \{t^\frown s \mid t^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(P), s \in \mathsf{divergences}(Q)\}\\
\mathsf{divergences}(P \setminus A) &= \{(t \upharpoonright (\Sigma \setminus A))^\frown r \mid t \in \mathsf{divergences}(P)\} \cup\\
&\quad\ \{(u \upharpoonright (\Sigma \setminus A))^\frown r \mid u \in \Sigma^\omega, u \upharpoonright (\Sigma \setminus A) \text{ finite}, \forall\, t < u \,\mathbf{.}\, t \in \mathsf{traces}_\perp(P)\}\\
\mathsf{divergences}(P[\![R]\!]) &= \{t^\frown r \mid \exists\, s \in \mathsf{divergences}(P) \cap \Sigma^* \,\mathbf{.}\, s\; R\; t\}\\
\mathsf{divergences}(P \underset{A}{\parallel} Q) &= \{u^\frown r \mid \exists\, s \in \mathsf{traces}_\perp(P), \exists\, t \in \mathsf{traces}_\perp(Q) \,\mathbf{.}\, u \in (s \underset{A}{\parallel} t \cap \Sigma^*),\\
&\quad\ (s \in \mathsf{divergences}(P) \text{ or } t \in \mathsf{divergences}(Q))\}
\end{aligned}$$

Figure 2.9: The model $\mathcal{N}$: inductive rules for calculating divergences

2.2.3.4 The Hierarchy

As we observed in the previous sections, given two CSP processes *Spec* and *Impl* and a CSP model $\mathcal{M} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$, the refinement check $\mathit{Spec} \sqsubseteq_{\mathcal{M}} \mathit{Impl}$ reduces to checking for reverse containment of possible behaviours. Formally:

$$\begin{aligned}
\mathit{Spec} \sqsubseteq_T \mathit{Impl} &\quad\longleftrightarrow\quad \mathsf{traces}(\mathit{Impl}) \subseteq \mathsf{traces}(\mathit{Spec})\\
\mathit{Spec} \sqsubseteq_F \mathit{Impl} &\quad\longleftrightarrow\quad \begin{array}{l}\mathsf{traces}(\mathit{Impl}) \subseteq \mathsf{traces}(\mathit{Spec}) \text{ and}\\ \mathsf{failures}(\mathit{Impl}) \subseteq \mathsf{failures}(\mathit{Spec})\end{array}\\
\mathit{Spec} \sqsubseteq_{FD} \mathit{Impl} &\quad\longleftrightarrow\quad \begin{array}{l}\mathsf{failures}_\perp(\mathit{Impl}) \subseteq \mathsf{failures}_\perp(\mathit{Spec}) \text{ and}\\ \mathsf{divergences}(\mathit{Impl}) \subseteq \mathsf{divergences}(\mathit{Spec})\end{array}
\end{aligned}$$

The three refinement relations are increasingly stronger: $\mathit{Spec} \sqsubseteq_{FD} \mathit{Impl}$ implies $\mathit{Spec} \sqsubseteq_F \mathit{Impl}$ and $\mathit{Spec} \sqsubseteq_F \mathit{Impl}$ implies $\mathit{Spec} \sqsubseteq_T \mathit{Impl}$. In addition, if a process *Impl* is divergence-free, then $\mathit{Spec} \sqsubseteq_{FD} \mathit{Impl}$ if and only if $\mathit{Spec} \sqsubseteq_F \mathit{Impl}$. Hence for divergence-free processes it is sufficient and a lot less computationally expensive to work in the stable-failures model.

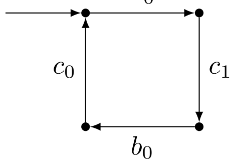
We remark that the three models that we described in the thesis are only a small fraction of all denotational models defined for CSP. A significant number of other more expressive CSP models are presented in [Ros11b]. Those include finite and infinite behavioural models, all of which study linear-time properties. A more general semantic-model hierarchy, analysing both linear and branching time models, can be found in [vG00, vG90]. The latter, however, has been compiled under the assumption that all actions are visible.

2.2.3.5 Calculating Denotational Values

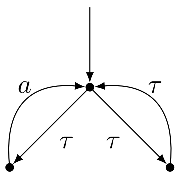
There are two different approaches for obtaining the denotational value $[\![P]\!]_M$ of a process P in a model $\mathcal{M}$—either by constructing it inductively from the denotational values of its subcomponents, or by extracting it from an operational semantics. To give a flavour, we presented some of the rules underlying the first approach in the previous sections. We refer the interested reader to [Ros98, Ros11b] for intuition about the rules, as well as for details about other semantic models of CSP. Since denotational values of processes are rather complex and often infinite, FDR calculates the behaviours of a process from its standard operational representation. This is justified by semantic models being congruent to the standard operational semantics. The congruence theorems are presented and proven in [Ros98].

2.2.4 Operational Semantics

The operational semantics allows us to associate to any CSP process a labelled transition systems (LTS) that represents its possible executions. The states of the transition systems represent process nodes and the labels—visible events or τ actions. For example, the LTS underlying the operational semantics of *Cell(0)* of Milner's scheduler is depicted in Figure 2.10(a). Figure 2.10(b) illustrates the operational representation of the process $P = (a \longrightarrow P \sqcap b \longrightarrow P) \setminus \{b\}$.



(a) LTS of Milner's *Cell(0)*



(b) LTS of P

Figure 2.10: Operational representation of CSP processes

2.2.4.1 SOS Rules

The operational semantics takes the form of a collection of SOS-style inference rules [Plo81, Plo04], called *firing rules*. Firing rules provide recipes for constructing an LTS out of a syntactic CSP description of a process. The recipes define how processes can evolve by calculating the initial actions available at each node in a single step and the possible results after performing each action.

The firing rules are presented below. We use an auxiliary process term Ω to denote any process that has already terminated successfully. If F is a CSP term with a free process variable X and Q is a CSP process, then $F[Q/X]$ represents the process obtained by substituting every free occurrence of X in F with Q. An occurrence of a variable X is free if it does not occur within the scope of a $\mu\, X$ operator. The last three rules reflect the fact that termination is distributive—$P_1 \underset{A}{\|} P_2$ terminates when both P_1 and P_2 do so. The reader is referred to [Ros98] for more information.

$$\frac{}{\mathit{SKIP} \xrightarrow{\checkmark} \Omega} \qquad (\mathit{SKIP})$$

$$\frac{}{x : A \to P(x) \xrightarrow{a} P(a)}(a \in A) \qquad (\xrightarrow{a})$$

$$\frac{}{P_1 \sqcap P_2 \xrightarrow{\tau} P_1} \qquad \frac{}{P_1 \sqcap P_2 \xrightarrow{\tau} P_2} \qquad (\sqcap)$$

$$\frac{}{\mu\, P \bullet F(P) \xrightarrow{\tau} F[(\mu\, P \bullet F(P))/P]} \qquad (\mu\, P \bullet F(P))$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \,\Box\, P_2 \xrightarrow{\tau} P_1' \,\Box\, P_2} \qquad \frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \,\Box\, P_2 \xrightarrow{\tau} P_1 \,\Box\, P_2'} \qquad (\Box_\tau)$$

$$\frac{P_1 \xrightarrow{b} P_1'}{P_1 \,\Box\, P_2 \xrightarrow{b} P_1'}(b \in \Sigma^{\checkmark}) \qquad \frac{P_2 \xrightarrow{b} P_2'}{P_1 \,\Box\, P_2 \xrightarrow{b} P_2'}(b \in \Sigma^{\checkmark}) \qquad (\Box_{\Sigma^{\checkmark}})$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \,\text{⨾}\, P_2 \xrightarrow{\tau} P_1' \,\text{⨾}\, P_2} \qquad (\text{⨾}_\tau)$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \mathbin{;} P_2 \xrightarrow{a} P_1' \mathbin{;} P_2}(a \in \Sigma) \qquad \frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1 \mathbin{;} P_2 \xrightarrow{\tau} P_2} \tag{$;_{\Sigma\checkmark}$}$$

$$\frac{P \xrightarrow{\tau} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \tag{$\setminus A_\tau$}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P \setminus A \xrightarrow{\checkmark} \Omega} \tag{$\setminus A_\checkmark$}$$

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}(a \in A) \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{a} P' \setminus A}(a \in \Sigma \setminus A) \tag{$\setminus A_\Sigma$}$$

$$\frac{P \xrightarrow{\tau} P'}{P[\![R]\!] \xrightarrow{\tau} P'[\![R]\!]} \tag{$[\![R]\!]_\tau$}$$

$$\frac{P \xrightarrow{\checkmark} P'}{P[\![R]\!] \xrightarrow{\checkmark} \Omega} \qquad \frac{P \xrightarrow{a} P'}{P[\![R]\!] \xrightarrow{b} P'[\![R]\!]}(a\ R\ b) \tag{$[\![R]\!]_{\Sigma\checkmark}$}$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} P_1' \underset{A}{\parallel} P_2} \qquad \frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} P_1 \underset{A}{\parallel} P_2'} \tag{$\underset{A}{\parallel}_\tau$}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1' \underset{A}{\parallel} P_2}(a \in \Sigma \setminus A) \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1 \underset{A}{\parallel} P_2'}(a \in \Sigma \setminus A) \tag{$\underset{A}{\parallel}_{\Sigma_1}$}$$

$$\frac{P_1 \xrightarrow{a} P_1',\ P_2 \xrightarrow{a} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{a} P_1' \underset{A}{\parallel} P_2'}(a \in A) \tag{$\underset{A}{\parallel}_{\Sigma_2}$}$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} \Omega \underset{A}{\parallel} P_2} \qquad \frac{P_2 \xrightarrow{\checkmark} P_2'}{P_1 \underset{A}{\parallel} P_2 \xrightarrow{\tau} P_1 \underset{A}{\parallel} \Omega} \tag{$\underset{A}{\parallel}_{\checkmark_1}$}$$

$$\frac{}{\Omega \underset{A}{\parallel} \Omega \xrightarrow{\checkmark} \Omega} \tag{$\underset{A}{\parallel}_{\checkmark_2}$}$$

2.2.4.2 Deriving Behaviours from Operational Semantics

We now present how traces, failures and divergences can be derived from the operational semantics of a process.

Let P be a finite-state process and $OS_P = \langle S^P, \mathsf{init}^P, A^P = \Sigma^{\tau,\checkmark}, T^P \rangle$ be the labelled transition system underlying the operational semantics of P.

We write $\Sigma^{*\checkmark}$ to denote the set of finite words over Σ which might end with $\checkmark$, and similarly for $(\Sigma^\tau)^{*\checkmark}$. Let $p, q \in S^P$, $w = \langle x_i \mid 0 \leq i < n \rangle \in (\Sigma^\tau)^{*\checkmark}$ and $t \in \Sigma^{*\checkmark}$.

Traces. In the following two definitions, we present two versions of the transitive closure of the transition relation T. The first one ($\longmapsto$) does not distinguish between visible and τ actions, whereas the second one ($\Longrightarrow$) ignores and trims τ actions in order to obtain a trace out of an execution word. Formally, we write $p \overset{w}{\longmapsto} q$ if there exists a sequence of states $\langle p_0, p_1, \ldots, p_n \rangle$, such that $p_0 = p$, $p_n = q$ and $p_k \xrightarrow{x_k} p_{k+1}$ for all $k \in \{0, \ldots, n-1\}$. We write $p \overset{t}{\Longrightarrow} q$ if there exists $w \in (\Sigma^\tau)^{*\checkmark}$, such that $p \overset{w}{\longmapsto} q$ and $t = w \upharpoonright \Sigma^\checkmark$. Then, we define the set of traces of the process P as follows:

$$\mathsf{traces}(P) = \{t \in \Sigma^{*\checkmark} \mid \exists\, q \in S^P \bullet \mathsf{init}^P \overset{t}{\Longrightarrow} q\}.$$

Stable Failures. We say that a state $p \in S^P$ is stable and write $\mathsf{stable}(p)$ if $p \overset{\tau}{\not\longrightarrow}$ and $p \overset{\checkmark}{\not\longrightarrow}$. We denote by $\mathsf{initials}(p)$ the set of all visible events that can be communicated from p, i.e., $\mathsf{initials}(p) = \{a \in \Sigma^\checkmark \mid p \xrightarrow{a}\}$.

For $p \in S^P$ and $X \subseteq \Sigma^\checkmark$, we say that p refuses X (alternatively, X is a refusal of s) and write $p \;\mathsf{ref}\; X$ if either of the following holds:

- $\mathsf{stable}(p)$ and $X \cap \mathsf{initials}(p) = \emptyset$, or
- $p \xrightarrow{\checkmark}$ and $X \subseteq \Sigma$.

Then, we define the set of stable failures of the process P as follows:

$$\begin{aligned} \mathsf{failures}(P) \quad = \quad & \{(t, X) \mid \exists\, q \in S^P \bullet \mathsf{init}^P \overset{t}{\Longrightarrow} q, q \;\mathsf{ref}\; X\} \cup \\ & \{(t^\frown\langle\checkmark\rangle, X) \mid X \subseteq \Sigma, \exists\, q \in S^P \bullet \mathsf{init}^P \xRightarrow{t^\frown\langle\checkmark\rangle} q\}. \end{aligned}$$

Note that refusal sets are subset closed: for any $X \subseteq \Sigma^\checkmark$, if $p \;\mathsf{ref}\; X$ and $Y \subseteq X$, then $p \;\mathsf{ref}\; Y$. Hence, we introduce a notion of a maximal refusal set. The set of events X is a maximal refusal of a state s if s refuses X and there is no proper superset Y of X with the same property. We define the set of all maximal refusal sets of p as follows:

$$\mathsf{max\_refusals}(p) = \{X \subseteq \Sigma^{\checkmark} \mid p \;\mathsf{ref}\; X, \nexists\, Y \subseteq \Sigma^{\checkmark} \bullet X \subsetneq Y \text{ and } p \;\mathsf{ref}\; Y\}.$$

In fact, instead of recording maximal refusal sets for each state, FDR follows a more space-efficient approach and stores their complements—the minimal acceptance sets. While maximal refusals specify what a process *may not* engage in, minimal acceptances specify what a process *must* perform if the environment desired.

$$\mathsf{min\_acceptances}(p) = \{Y \subseteq \Sigma^{\checkmark} \mid \exists\, X \subseteq \Sigma^{\checkmark} \bullet X \in \mathsf{max\_refusals}(p), Y = \Sigma^{\checkmark} \setminus X\}.$$

Deadlock, Nondeterminism, Livelock. We say that a state $p \in S^P$ is divergent and write $p \Uparrow$ if there exists an infinite sequence of states $\langle p_0, p_1, p_2, \ldots \rangle$, such that $p_0 = p$ and, for all $n \in \mathbb{N}$, $p_n \xrightarrow{\tau} p_{n+1}$. We can define the set of divergences of the process P as follows:

$$\mathsf{divergences}(P) = \{t^\frown t' \mid t \in \Sigma^*, t' \in \Sigma^{*\checkmark}, \exists\, q \in S^P \bullet \mathsf{init}^P \overset{t}{\Longrightarrow} q \text{ and } q \Uparrow\}.$$

A process P is then divergent if $\mathsf{divergences}(P) \neq \emptyset$. P can deadlock if there exists $t \in \mathsf{traces}(P)$, such that $(t, \Sigma) \in \mathsf{failures}(P)$. P is nondeterministic if, either it is divergent, or there exist $t \in \mathsf{traces}(P)$ and $a \in \Sigma$, such that $t^\frown\langle a \rangle \in \mathsf{traces}(P)$ and $(t, \{a\}) \in \mathsf{failures}(P)$.

2.2.4.3 Bisimulation Orders

Since the LTS representation of a process is not unique, in terms of operational semantics, two processes are considered equivalent if their transition systems are bisimilar. In this section, we describe several alternatives of bisimulation relations—strong (standard for CSP) [Ros98], weak (observational) [Ros98] and divergence-respecting weak bisimulation [Ros11b], all of which are equivalence relations (see Section 2.1.3.2). Intuitively, two transition systems are bisimilar if they can mimic (simulate) each other stepwise with respect to certain criteria, i.e., if they can initially and subsequently exhibit identical behaviours.

In general, bisimulation plays a central role in process algebras [Par81, Mil89]. Checking whether two processes are equivalent in a given semantic model reduces to checking for behavioural language equivalence which is PSPACE-complete [KS83]. Bisimulation equivalence, on the other hand, is in PTIME and implies language equivalence (the converse does not necessarily hold). Two strongly-bisimilar or DRW-bisimilar processes have precisely the same sets of behaviours in all three CSP models $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{FD}$ [Ros11b]. Weak bisimulation is only relevant in the traces model $\mathcal{T}$ as it fails to distinguish between $STOP$ and DIV and between stable and unstable states, in general [Ros11b]. Hence, a sound but incomplete method for establishing semantic process equivalence in CSP is by identifying a strong or a DRW bisimulation equivalence between the corresponding transition systems. In addition, if S strongly or divergence-respecting weakly simulates I, then $S \sqsubseteq_{\mathcal{M}} I$ for any $\mathcal{M} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$.

Let throughout this section $M = \langle S, \mathsf{init}, \Sigma^{\tau,\checkmark}, T\rangle$ and $M_i = \langle S_i, \mathsf{init}_i, \Sigma^{\tau,\checkmark}, T_i\rangle$, for $i = 1, 2$, be labelled transition systems over a set of actions $\Sigma^{\tau,\checkmark}$. A *strong bisimulation* for M is a binary relation R on S satisfying the following requirement:

1. For any $(s_1, s_2) \in R$ and any $x \in \Sigma^{\tau,\checkmark}$:
   - if $s_1 \xrightarrow{x} s_1'$, then there exists $s_2' \in S$ such that $s_2 \xrightarrow{x} s_2'$ and $(s_1', s_2') \in R$, and
   - if $s_2 \xrightarrow{x} s_2'$, then there exists $s_1' \in S$ such that $s_1 \xrightarrow{x} s_1'$ and $(s_1', s_2') \in R$.

States s_1 and s_2 of M are *strongly bisimilar* if there exists a strong bisimulation relation R for M such that $(s_1, s_2) \in R$. Strong bisimulation is an equivalence relation and a coarsest strong bisimulation for M exists. The state space of M can be reduced without changing the semantic value of the underlying process by quotienting M with respect to the coarsest strong bisimulation relation. As there is a bijection between equivalence relations and partitions of the state space (and between classes of equivalence and blocks in a partition, see Section 2.1.3.2), the algorithm is based on iterative partition refinement by repeatedly splitting blocks according to certain criteria until a fixed point is reached [CGP99, BK08]. Equivalently, a logical characterisation of the coarsest bisimulation equivalence can be given as the greatest fixed point of a function over binary relations [Mil89].

Strong bisimulation can be lifted from an equivalence relation $R \subseteq S \times S$ between the states of a transition system M to an equivalence relation $R \subseteq S_1 \times S_2$ between two different transition systems M_1 and M_2 by further requiring that the initial states of M_1 and M_2 are bisimilar. Formally, a strong bisimulation for (M_1, M_2) is a binary relation $R \subseteq S_1 \times S_2$ satisfying the following requirements:

1. $(\mathsf{init}_1, \mathsf{init}_2) \in R$.
2. For any $(s_1, s_2) \in R$ and any $x \in \Sigma^{\tau,\checkmark}$:
   - if $s_1 \xrightarrow{x}_1 s_1'$, then there exists $s_2' \in S_2$ such that $s_2 \xrightarrow{x}_2 s_2'$ and $(s_1', s_2') \in R$, and
   - if $s_2 \xrightarrow{x}_2 s_2'$, then there exists $s_1' \in S_1$ such that $s_1 \xrightarrow{x}_1 s_1'$ and $(s_1', s_2') \in R$,.

Furthermore, M_1 and M_2 are strongly bisimilar if there exists a strong bisimulation R for (M_1, M_2).

In general, strong bisimulation does not distinguish between invisible τ actions and visible events when declaring two states equivalent. Weak bisimulation, on the other hand, does make this distinction and considers two process states equivalent if they offer the same sets of single-event traces (provided that those traces lead to successor states that are themselves bisimilar). Weak bisimulation is refined by DRW bisimulation by further requiring that two equivalent states are either both immediately divergent or both immediately non-divergent. In fact, strong, DRW and weak bisimulation are increasingly coarser relations—every strong bisimulation is a DRW bisimulation and every DRW bisimulation is a weak bisimulation.

Formally, a *divergence-respecting weak bisimulation* for M is a binary relation R on S satisfying the following requirements:

1. If $(s_1, s_2) \in R$, then $s_1 \Uparrow$ if and only if $s_2 \Uparrow$.
2. For any $(s_1, s_2) \in R$ and any $t \in \Sigma^{*,\checkmark}$:
   - if $s_1 \overset{t}{\Longrightarrow} s_1'$, then there exists $s_2' \in S$ such that $s_2 \overset{t}{\Longrightarrow} s_2'$ and $(s_1', s_2') \in R$, and
   - if $s_2 \overset{t}{\Longrightarrow} s_2'$, then there exists $s_1' \in S$ such that $s_1 \overset{t}{\Longrightarrow} s_1'$ and $(s_1', s_2') \in R$.

A *weak bisimulation* for M can be obtained from the definition of DRW bisimulation above by dropping the first requirement. Similarly to strong bisimulation, weak and DRW bisimulations are equivalence relations and can be employed in semantic-preserving state-space compression techniques (only within the traces model in case of weak bisimulation). Furthermore, both bisimulation relations can also be lifted to binary relations on transition systems.

2.3 FDR

2.3.1 The Two-Level Operational Semantics

The SOS notation for operational semantics allows the creation of many operators that do not fit in the denotational world of CSP. Any CSP operator can be described using less general *combinator*-style rules instead and, conversely, any operator that can be given combinator-style operational semantics can be derived and given denotational semantics in CSP [Ros11b]. Combinator-style operational semantics can be generalised to *supercombinator*-style operational semantics which is the one used in FDR. We give details about both combinator and supercombinator semantics below.

2.3.1.1 Combinator-Style Operational Semantics

As with SOS, there are several combinator rules for each CSP operator and these allow us to infer the initial actions available at each process node out of its top-level operator and the initial actions available at its immediate process arguments. The crucial difference compared to SOS rules originates from the fact that process arguments can be viewed as switched on or off, depending on the context they are used in.

Switched-On and Switched-Off Process Arguments. Given a compound CSP process $P = \otimes(P_1, \ldots, P_n)$, a process argument P_i is considered *switched on* if its initial actions are immediately relevant for the initial actions of P and *switched off* if $\otimes$ does not need its initial actions to deduce the resulting initial action of P.

Example 2.3.1.

1. *In* $P_1 \mathbin{|||} P_2 = P_1 \underset{\emptyset}{\parallel} P_2$*, both* P_1 *and* P_2 *are switched on*
2. *In* $P_1 \mathbin{⨟} P_2$*,* P_2 *is initially switched off until* P_1 *performs* $\checkmark$*, at which point* P_1 *becomes switched off and* P_2 *switched on*
3. *In* $a \longrightarrow P$*,* P *is initially switched off but gets switched on when* a *is communicated*
4. *In* $P_1 \sqcap P_2$*,* P_1 *and* P_2 *are initially switched off as the nondeterministic choice is only resolved after a* τ *is performed, at which point precisely one of the two processes becomes activated.*

Types of Combinator Rules. Combinator rules keep track of which processes are switched on at every given moment and restrict SOS by allowing only two types of rules:

- *τ-promotion rules*: those rules enforce that whenever a switched-on process argument performs a τ, this is promoted to a τ of the compound process that does not change its structure.
- *Visible-event rules*: those rules combine visible events of switched-on process arguments (if any) into a resulting action of the compound process. In those rules, a switched-on process can participate with either a visible event or not be involved at all, the latter of which we denote with the symbol ϵ.

Structure of Successor Terms. Combinator rules also need to indicate the structure of the successor term. In many instances, the structure is the same as the initial one and so does not have to be mentioned explicitly in the rules. When the structure does change (i.e., processes become switched from on to off or conversely), this is indicated by a CSP term in which the various arguments of the operator may appear. In any case, the successor state contains the original argument if the the latter has not participated in the action, or the state that the argument has moved to if it did.

Now formally [Ros11a], let P be a compound process with a top-level operator $\otimes$, switched-on arguments $P_1, \ldots, P_n$ (for some $n \geq 0$) and switched-off arguments $Q = \langle P_\lambda \mid \lambda \in \Lambda \rangle$.

Having any switched-on process argument P_i that can go via a τ to a state P_i', the τ-promotion rule takes the form:

$$\otimes(P_1, \ldots, P_i, \ldots P_n, Q) \xmapsto{\tau} \otimes(P_1, \ldots, P_i', \ldots P_n, Q).$$

As this rule holds universally for any switched-on argument of any CSP operator, τ-promotion rules do not need to be added explicitly to the combinator operational semantics as they were in the SOS rules $\Box_\tau, ⨾_\tau, \backslash A_\tau, [\![R]\!]_\tau, \underset{A}{\|}\,_\tau$.

Rules combining visible events take the general form $((x_1, \ldots, x_n), y, T)$, where $x_i \in \Sigma^{\checkmark} \cup \{\epsilon\}$, $y \in \Sigma^{\tau,\checkmark}$ and T is a piece of CSP syntax specifying the structure of the successor term. The idea is that whenever all P_i's that have $x_i \neq \epsilon$ can perform x_i and go to states

P_i', they can synchronise to make the compound process P perform y and enter a state T. The successor state T is either Ω, if $y = \checkmark$, or is specified by an open CSP term in which the free variables are indices drawn from $\{1, \ldots, n\} \cup \{\text{-}\lambda \mid \lambda \in \Lambda\}$, which get substituted according to the following rules:

- For $i \in \{1, \ldots, n\}$, we distinguish different cases. If $x_i = \epsilon$ or $x_i \in \Sigma$, i is replaced by P_i or P_i', respectively. If $x_i = \checkmark$, i does not appear in the successor term T any more as P_i becomes switched off.
- An index -λ for $\lambda \in \Lambda$ indicates that the process P_λ has become switched on and is replaced by P_λ.

Combinator Rules. We list the combinator rules below. In some of them, e.g., $\fatsemi_\Sigma$, $\backslash A_\Sigma$, $[\![R]\!]_\Sigma$, $\underset{A}{\parallel}_{\checkmark 1}$ and $\underset{A}{\parallel}_{\checkmark 2}$, the structure of the successor term does not change, i.e., the resulting state is $\otimes(P_1'', \ldots, P_n'', Q)$, where $P_i'' = P_i$ if $x_i = \epsilon$ and $P_i'' = P_i'$ if $x_i \in \Sigma$. In those cases, we omit T from the rules for simplicity. In rules *SKIP*, $\xrightarrow{a}$ and $\sqcap$, there is no switched-on argument initially which we indicate by $-$. Ω is naturally switched off as it represents successful termination. Hence, $\Omega \underset{A}{\parallel}$ and $\underset{A}{\parallel} \Omega$ are viewed as unary operators.

$$((-), \checkmark, \Omega) \qquad (SKIP)$$

$$((-), a, \text{-}1) \qquad (\xrightarrow{a})$$

$$((-), \tau, \text{-}1) \text{ and } ((-), \tau, \text{-}2) \qquad (\sqcap)$$

$$((\checkmark, \epsilon), \checkmark, \Omega) \text{ and } ((\epsilon, \checkmark), \checkmark, \Omega) \qquad (\Box_\checkmark)$$

$$((a, \epsilon), a, 1) \text{ and } ((\epsilon, a), a, 2) \qquad (\Box_\Sigma)$$

$$((a), a) \text{ and } ((\checkmark), \tau, \text{-}2) \qquad (\fatsemi_{\Sigma\checkmark})$$

$$((\checkmark), \checkmark, \Omega) \qquad (\backslash A_{\checkmark})$$

$$((a), \tau) \text{ if } a \in A \text{ and } ((a), a) \text{ if } a \in \Sigma \setminus A \qquad (\backslash A_{\Sigma})$$

$$((\checkmark), \checkmark, \Omega) \text{ and } ((a), b) \text{ when } a \mathrel{R} b \qquad ([\![R]\!]_{\Sigma\checkmark})$$

$$((a, \epsilon), a) \text{ and } ((\epsilon, a), a) \text{ if } a \in \Sigma \setminus A \qquad (\underset{A}{\|}\ {}_{\Sigma_1})$$

$$((a, a), a) \text{ if } a \in A \qquad (\underset{A}{\|}\ {}_{\Sigma_2})$$

$$((\checkmark, \epsilon), \tau, \Omega \underset{A}{\|} 2) \text{ and } ((\epsilon, \checkmark), \tau, 1 \underset{A}{\|} \Omega) \qquad (\underset{A}{\|}\ {}_{\checkmark_1})$$

$$((a), a) \text{ if } a \in \Sigma \setminus A \text{ and } ((\checkmark), \tau, \mathit{SKIP}) \qquad (\Omega \underset{A}{\|}, \underset{A}{\|} \Omega)$$

2.3.1.2 Supercombinator-Style Operational Semantics

Combinator-style operational semantics captures precisely CSP-definable operators [Ros11b], [Ros11a]. However, actions of compound processes need to be calculated recursively on-the-fly out of the actions of subterms. Furthermore, successor states are presented as pieces of syntax which does not prove to be efficient when analysing large systems.

From Combinators to Supercombinators. Supercombinator-style operational semantics is a less general but more efficient version of the combinator-style operational semantics [Ros11b]. Supercombinator rules take the form of combinator ones, but are generalised to combine together actions of subprocesses nested under an *arbitrary* number of applications of CSP operators. As there is no combinator rule for recursion, the only constraint is that any process argument should be a closed CSP term, i.e., should have all the recursion unwound. Based on this assumption, combinator rules of process arguments can be composed together to obtain rules for the outermost CSP operators. In this way, a combination of

CSP operators can be transformed into a single CSP operator, flattening the multiple layers of nesting. Furthermore, this can be implemented efficiently in a single run before the state-space exploration phase rather than on-the-fly every time when necessary via recursive calls.

We illustrate the approach by example. Let $P = \otimes_1(\otimes_2(P_1, P_2), \otimes_3(P_3, P_4))$, where $\otimes_1, \otimes_2$ and $\otimes_3$ are binary CSP operators and P_1, P_2, P_3 and P_4 are CSP processes. We want to derive supercombinator rules for a quaternary operator $\otimes$ such that $\otimes(P_1, P_2, P_3, P_4) = \otimes_1(\otimes_2(P_1, P_2), \otimes_3(P_3, P_4)) = P$. Let us assume for simplicity that $\otimes_1, \otimes_2$ and $\otimes_3$ have all their arguments switched on and their application does not change the structure of the successor terms. Considering the τ-promotion rules, if $\otimes_2$ or $\otimes_3$ have a rule that generates a τ, this τ gets promoted by $\otimes_1$ to $\otimes$. For instance, if $\otimes_2$ has the rule $((a, b), \tau)$, then we create a supercombinator rule $((a, b, \epsilon, \epsilon), \tau)$ for the compound operator $\otimes$. The other type of supercombinator rule arises when we can match all input requirements of one of $\otimes_1$'s combinators using combinators of $\otimes_2$ and $\otimes_3$ that produce visible results. For example, if $\otimes_1, \otimes_2$ and $\otimes_3$ have the rules $((a, b), c)$, $((\epsilon, a), a)$ and $((b, d), b)$, respectively, then the compound operator $\otimes$ will have the rule $((\epsilon, a, b, d), c)$.

Supercompilation. The process of associating to any CSP process supercombinator-style operational semantics is called *supercompiling* and it follows a hybrid high-/low-level approach for calculation and representation [Ros11b]. It identifies all true recursions and compiles them on a low level, generating explicit LTS's using combinator rules. What remains for the high level are closed CSP terms (i.e., CSP processes) combined typically using parallel composition, hiding and renaming, although the dividing line is somewhat more complex and can be drawn where sensible. For example, the choice operators and sequential composition can also be lifted to the high level as long as their arguments are all closed terms.

The result of supercompilation is a high-level structure which consists of two parts.

1. *A process tree* with leaves—low-level compiled LTS's, and internal nodes—CSP operators compiled on the high level, usually hiding, renaming or parallel composition.

2. *A set of supercombinators* mapping actions of a number of leaf processes to an event-outcome of the composite root process [Ros98]. In what follows, we use the notions of supercombinators and rules interchangeably.

We note that the list of leaf processes together with the set of supercombinators is a complete characterisation of the high-level process as the semantics of all CSP operators corresponding to the internal nodes in the process tree is captured by the supercombinator rules. The structured process tree can be used, though, for making the whole high-level process completely explicit, as well as for debugging purposes.

Supercombinator Rules and Formats. The set of combinators is partitioned with respect to the existing *formats*—the different configurations of switched-on and switched-off leaf processes. In the worst case, the number of formats can be exponential in the number of leaves, but in practice this is rarely the case and quite often, there is just a single format, especially when composing processes in parallel and abstracting away from certain details on top, which is a very common pattern in practice.

Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all, the latter of which we again denote by ϵ. As with combinator rules, the supercompiler generates two types of rules [Ros98, Gol04, RRS+01]:

1. A rule for a leaf process willing to perform a τ which promotes a τ action of the root process.
2. Rules using visible actions.

Note that the visible actions that the leaf processes perform need not be the same if hiding or renaming is involved in the combination being modelled. For example, if $P = a \longrightarrow P$ and $Q = b \longrightarrow Q$, then if P performs a and Q performs b, $P \underset{\{a\}}{\|} Q[\![\mathrm{a}/b]\!]$ can perform a, where $Q[\![\mathrm{a}/b]\!]$ is the process Q with the event b being renamed to a. Hence, $((a, b), a)$ is a valid rule for the root process $P \underset{\{a\}}{\|} Q[\![\mathrm{a}/b]\!]$ with leaves P and Q.

Example 2.3.2. *Going back to our running example, after supercompiling Scheduler* $\setminus \{|\, b \,|\}$*, we obtain the process tree depicted in Figure 2.11. The simple recursive cell processes are compiled as leaves and their LTS's generated explicitly. The root process contains just a single format with three types of supercombinators:*

1. *If Cell(i) and Cell$(i \oplus 1)$ perform c.$i \oplus 1$, Scheduler $\setminus \{| b |\}$ performs a τ*
2. *If Cell(i) performs a.i, Scheduler $\setminus \{| b |\}$ also performs a.i*
3. *If Cell(i) performs b.i, Scheduler $\setminus \{| b |\}$ performs a τ*

$$\begin{tikzcd}
 & & & \setminus \{| b |\} \arrow[d] & & & & \\
 & & & \setminus \{| c |\} \arrow[d] & & & & \\
 & & & \| \arrow[dll] \arrow[drr] & & & & \\
 & \| \arrow[dl] \arrow[dr] & & & & \| \arrow[dl] \arrow[dr] & & \\
 \mathit{Cell(0)} & & \mathit{Cell(1)} & & \mathit{Cell(2)} & & \| \arrow[dl] \arrow[dr] & \\
 & & & & & \mathit{Cell(3)} & & \mathit{Cell(4)}
\end{tikzcd}$$

Figure 2.11: Process tree for *Scheduler* $\setminus \{| b |\}$

Supercombinator operational representation can be considered an implicit LTS because it gives an initial state and sufficient information to calculate all the transitions of the system on-the-fly. Given a root high-level process, we refer to tuples of the current states of its leaf processes as *configurations*. When running the root process, FDR computes its initial actions by checking which supercombinators are enabled from the current configuration and the current format of the root. A supercombinator might be disabled if not all leaf processes are currently able to communicate the event that they are responsible for within the supercombinator. Hence, the operational semantics of the root process can be considered an implicit LTS, whose transitions can be switched on and off. The states are represented by a pair of a configuration and a format of the root. Transitions are modelled by supercombinators.

Example 2.3.3. *Going back to Milner's scheduler, the supercombinator $((c.1, c.1, \epsilon, \epsilon, \epsilon), \tau)$ (see Figure 2.12) would be enabled if Cell(0) is in state 1 and Cell(1) is in state 0, regardless of the current states of the other three cell processes. If this rule is enabled and the transition taken, Cell(0) will go to state 2, Cell(1) will go to state 1, the other three cell processes will not progress and Scheduler $\setminus \{| b |\}$ will perform a τ.*

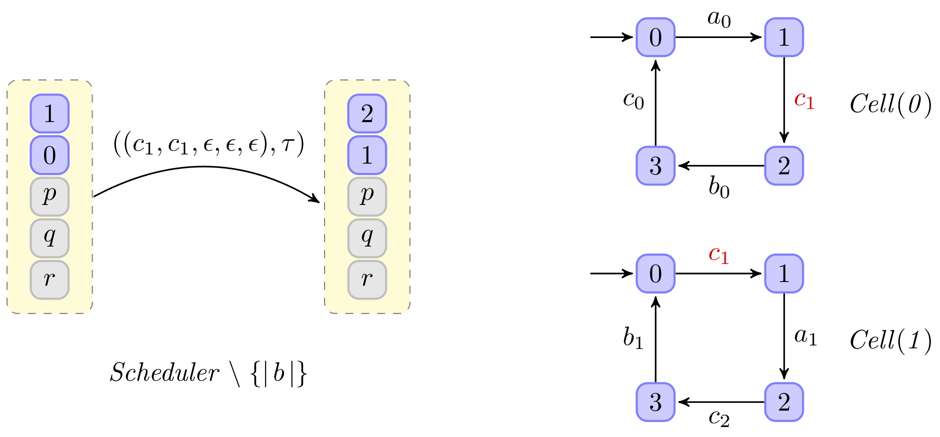

1
0
p
q
r
$((c_1, c_1, \epsilon, \epsilon, \epsilon), \tau)$
2
1
p
q
r
Scheduler $\setminus \{| b |\}$
a_0
c_0
c_1
b_0
Cell(0)
c_1
b_1
a_1
c_2
Cell(1)

Figure 2.12: Operational semantics of a high-level process

To summarise, supercombinators can be viewed as implicit state-space representations. They are generated by mimicking the SOS or combinator rules, but yield more compact storage and more efficient algorithms. Therefore, FDR is the most efficient when manipulating processes with relatively simple sequential leaves composed in parallel or applied hiding or renaming upon. Of course, high-level processes can be explicated, i.e., transformed into explicit LTS's, paying a potentially exponential price. This is quite logical as explication breaks down the hierarchical structure of a system composed of concurrent processes and makes it sequential.

2.3.2 Refinement Checking

FDR carries out the refinement check on the level of the LTS representations $OS_{Spec} = \langle S^s, \mathsf{init}^s, L^s, T^s \rangle$ and $OS_{Impl} = \langle S^i, \mathsf{init}^i, L^i, T^i \rangle$ of the specification and the implementation, respectively.

The algorithm is similar to the standard one for deciding language containment $L(\mathcal{A}) \subseteq L(\mathcal{B})$ of nondeterministic automata $\mathcal{A}$ and $\mathcal{B}$, which reduces to checking whether $L(\mathcal{A}) \cap \overline{L(\mathcal{B})} = \emptyset$ and requires that $\mathcal{B}$ be *a priori* transformed into a deterministic automaton.

Normalisation of the Specification. In a similar fashion, as a preprocessing step, FDR normalises OS_{Spec}, so that OS_{Spec} reaches a unique state after any trace. The normalisation procedure requires as a precondition that OS_{Spec} be explicated and therefore *Spec* sequentialised. Essentially, the normalisation procedure transforms OS_{Spec} into the unique semantically-equivalent τ-free deterministic bisimulation-reduced LTS. The algorithm for normalisation is very similar to the subset construction used for determinising automata and, consequently, a state of the normalised OS_{Spec} usually corresponds to a set of states of the original OS_{Spec}. Therefore, normalisation can potentially incur an exponential blow-up of the state space, although in practice this is rarely the case.

Generalised Transition Systems. In general, state-space compression techniques, an example of which is normalisation, group together sets of states into multistates. As a result, if working in the stable-failures or in the failures-divergences models, information about refusals and divergences can become lost or distorted. In order to preserve the τ-related nondeterminism of multistates, after compression the operational semantics of a process becomes a *generalised LTS* (GLTS) [Ros98], with each (multi) state s being additionally annotated with:

1. An antichain of minimal acceptances $\mathsf{min\_acceptances}(s)$: a set of sets of events that are incomparable under the subset order. If s is stable, then $\mathsf{min\_acceptances}(s)$ is not empty and every $A \in \mathsf{min\_acceptances}(s)$ is a subset of the visible events that s can stably communicate.
2. A flag $\delta(s)$ indicating whether s is divergent.

In general, a state of the GLTS can still have ambiguous branching under visible events, just as a standard LTS.

State-Space Exploration. After normalising OS_{Spec}, FDR traverses the Cartesian product of OS_{Spec} and OS_{Impl} in a breadth-first manner, checking for compatibility of mutually-reachable states. For the failures-divergences model, a pair of states (s^s, s^i) is compatible if the following conditions hold:

- For any $a \in \Sigma^{\checkmark}$, if $s^i \xrightarrow{a}$, then $s^s \xrightarrow{a}$, i.e., $\mathsf{initials}(s^i) \subseteq \mathsf{initials}(s^s)$.
- For any $X \subseteq \Sigma^{\checkmark}$, if $s^i \mathrel{\mathsf{ref}} X$, then $s^s \mathrel{\mathsf{ref}} X$, i.e.,
  for any $X \subseteq \Sigma^{\checkmark}$, if $X \in \mathsf{max\_refusals}(s^i)$, then $s^s \mathrel{\mathsf{ref}} X$, i.e.,
  for any $X \in \mathsf{min\_acceptances}(s^i)$, there exists $Y \in \mathsf{min\_acceptances}(s^s)$, such that $Y \subseteq X$.

- If s^i is divergent, then s^s is divergent, i.e., $s^i \Uparrow$ implies $s^s \Uparrow$.

Algorithm 1 Refinement checking in FDR [RRS+01]

```
push(Explorer, (init^s, init^i))
while Explorer not empty do
    $(s^s, s^i) = pop(Explorer)$
    if $\mathsf{compatible}(s^s, s^i, M)$ then
        $NewStates = \{(t^s, t^i) \mid \exists\, a \in \Sigma^s \cap \Sigma^i \bullet s^s \xrightarrow{a} t^s \text{ and } s^i \xrightarrow{a} t^i\} \cup$
                     $\{(t^s, t^i) \mid s^s = t^s \text{ and } s^i \xrightarrow{\tau} t^i\}$
        for all $(t^s, t^i) \in NewStates$ do
            if $(t^s, t^i)$ has not been explored yet then
                $push(Explorer, (t^s, t^i))$
            end if
        end for
    else
        return SPECIFICATION VIOLATED and a counterexample behaviour
    end if
end while
return SPECIFICATION HOLDS
```

The refinement algorithm is sketched as Algorithm 1. The compatibility of s^s and s^i in a model M is defined depending on M as follows:

- $\mathsf{compatible}(s^s, s^i, \mathcal{T})$ iff $\mathsf{initials}(s^i) \subseteq \mathsf{initials}(s^s)$
- $\mathsf{compatible}(s^s, s^i, \mathcal{F})$ iff $\mathsf{compatible}(s^s, s^i, \mathcal{T})$ and, if $\mathsf{stable}(s^i)$, then for any $X \in \mathsf{min\_acceptances}(s^i)$, there exists $Y \in \mathsf{min\_acceptances}(s^s)$, such that $Y \subseteq X$
- $\mathsf{compatible}(s^s, s^i, \mathcal{N})$ iff $\mathsf{compatible}(s^s, s^i, \mathcal{F})$ and $s^i \Uparrow$ implies $s^s \Uparrow$

If the property is violated, the breadth-first mode of search guarantees that that the counterexample generated is of minimal length.

We remark that for the failures-divergences model, FDR actually exploits a hybrid BFS/DFS strategy. The main exploration of the state space is performed in BFS mode, but in order to check for divergence, a local DFS search is triggered each time when encountering an unstable state. Therefore, despite employing hashing techniques, the refinement check in the failures-divergences model is much more computationally expensive than the one in the stable-failures model.

2.3.3 Abstraction and Compression Techniques

2.3.3.1 Semantic-Preserving Transformations on Transition Systems

To combat the state-space explosion problem, FDR provides several compression functions that can be used for transforming the LTS of a process into a semantically equivalent LTS with a smaller state space. We give a short description of several compression functions below. All of them preserve the semantic value of the process in all three CSP models $\mathcal{T}, \mathcal{F}$ and $\mathcal{N}$.

Normalisation. Normalisation is automatically applied on the left-hand side of a refinement check, but it can also be used as a state-space compression technique in its own right. First the LTS is transformed into a deterministic τ-free GLTS in pre-normal form, as described in Section 2.3.2. Pre-normalisation essentially eliminates all nondeterminism, guaranteeing that a unique state is reached after following any given trace, the cost being a potentially exponential increase in the state space. The true normal form is obtained by computing and quotienting the pre-normal form by its coarsest strong-bisimulation relation, as described in Section 2.2.4.3.

Strong-bisimulation reduction. This compression function factors a GLTS by its coarsest strong-bisimulation relation and is applied automatically to every leaf process in the final stage of supercompilation. It is also frequently applied after carrying out diamond elimination, which we describe below. Strong-bisimulation reduction never expands the state space of a system.

DRW-bisimulation reduction. Similarly to the previous one, this compression function factors a GLTS by its coarsest DRW-bisimulation relation. Typically, it does not achieve the same degree of compression as the combination of diamond elimination and strong-bisimulation reduction, but has the great advantage of preserving semantic values in all denotational models of CSP, not only the ones presented in this thesis.

τ-loop elimination. This compression technique is based on the fact that states that are mutually τ-reachable from each other exhibit exactly the same behaviours in all three models $\mathcal{T}, \mathcal{F}$ and $\mathcal{N}$. Hence, τ-loop elimination compresses an LTS or a GLTS by factoring it by an equivalence relation that considers all states in a strongly τ-connected component equivalent. This equivalence relation is a DRW bisimulation although not necessarily the coarsest one [Ros11b]. The resulting GLTS has the property that τ-reachability is a strict partial order on its state space.

Diamond elimination. Diamond elimination requires as a precondition that τ-reachability be a partial order on the states of the transition system and is usually applied on top of τ-loop elimination, which provides this guarantee (unless the process is known to be livelock-free, in which case the property holds automatically). Diamond elimination transforms a GLTS into a τ-free GLTS of size not greater than the original one, where there might still be nondeterminism left as far as visible events are concerned. Hence this compression technique is beneficial only when applied to transition systems with τ actions.

For nondeterministic processes, it is usually hard to formulate strict rules and predict which compression function would prove beneficial for a certain CSP process or refinement check. For deterministic processes, all three of DRW-bisimulation reduction, normalisation and the composition of strong-bisimulation reduction and diamond elimination yield isomorphic transition systems [Ros11b].

2.3.3.2 Hierarchical Compression

Since the main source of state-space explosion is concurrency, compression functions are usually applied on partially constructed systems, e.g., on sequential processes before composing them in parallel, or on subsystems that have been already compressed. FDR also supports a number of higher-order compression functions that apply a selected compression function automatically and inductively on subcomponents during the process of constructing a system. The latter technique is known as *hierarchical compression*, two examples of which we briefly describe below, namely *leaf compression* and *inductive compression.*

A vast majority of CSP implementations are constructed by taking a family of alphabetised processes $\langle (P_i, A_i) \mid i \in I \rangle$, composing them in parallel and abstracting away a set of events H at the top:

$$\mathit{System} = (\ \Vert_{i \in I} A_i \ @\ P_i) \setminus H.$$

This construction uses the replicated alphabetised parallel operator which requires that each process P_i come equipped with its alphabet of events A_i. Both leaf compression and inductive compression aim at compressing CSP processes that adhere to this template. In the notation below, let for each $i \in I$, $L_i \subseteq A_i$ be the set of events that are local for P_i, i.e., the largest subset of A_i such that for all $j \neq i$, $L_i \cap A_j = \emptyset$. We generalise this by denoting by $\mathsf{local}(Q)$ the set of events that are local for a process Q. Let $\mathsf{compress}$ be any of the compression functions that we listed in Section 2.3.3.1 above. Both leaf compression and inductive compression take the following arguments:

1. A semantic-preserving LTS compression function $\mathsf{compress}$,
2. A list of alphabetised processes $PAList = \langle (P_i, A_i) \mid i \in I \rangle$, and
3. A set of events H to hide on top level.

Then, the functions

$$\mathsf{LeafCompress}(\mathsf{compress})(PAList)(H) = (\ \big\Vert_{i \in I} A_i - L_i \ @\ \mathsf{compress}(P_i \setminus L_i)) \setminus (H - \bigcup_{i \in I} L_i)$$

and $\mathsf{InductiveCompress}(\mathsf{compress})(PAList)(H)$ follow, respectively, Algorithms 2 and 3 below and produce processes semantically equivalent to and not larger than *System*.

Algorithm 2 $\mathsf{LeafCompress}(\mathsf{compress})(\langle (P_i, A_i) \mid i \in I \rangle)(H)$

```
1: for $i \in I$ do
2:     compute the local events $L_i \subseteq A_i$ of $P_i$
3:     hide the local events $L_i$ in $P_i$
4:     $\mathsf{compress}(P_i \setminus L_i)$
5: end for
6:
7: // Compose the compressed processes in parallel
8: $Network = \ \big\Vert_{i \in I} A_i - L_i \ @\ \mathsf{compress}(P_i \setminus L_i)$
9:
10: // Hide the remaining events from $H$ on top and return
11: return $Network \setminus (H - \bigcup_{i \in I} L_i)$
```

Leaf compression is beneficial whenever leaf processes have a significant number of actions that can be hidden and that are local, i.e., not used for synchronisation with other components. The same holds for inductive compression, where the effect is amplified at each iteration because events that are not local for any process P_i can become local for the part of the network that has been already constructed. We note that for inductive compression the order of arranging the processes in *PAList* is important.

Algorithm 3 InductiveCompress(compress)$(\langle (P_i, A_i) \mid i \in I \rangle)(H)$

```
1: let $I = \{1, \ldots, n\}$
2: compute the local events $L_n \subseteq A_n$ of $P_n$
3: compress$(P_n \setminus L_n)$
4: $Network$ = compress$(P_n \setminus L_n)$
5:
6: **for** $i = n - 1 \rightarrow 1$ **do**
7:     compute the local events $L_i \subseteq A_i$ of $P_i$
8:     compress$(P_i \setminus L_i)$
9:     // compose in parallel with the network built so far
10:    $Network'$ = compress$(P_i \setminus L_i) \parallel Network$
11:    compute local$(Network')$
12:    $Network$ = compress$(Network' \setminus$ local$(Network'))$
13:    $i = i - 1$
14: **end for**
15:
16: // Hide the remaining events from $H$ on top and return
17: return $Network \setminus H$
```

Chapter 3

SAT-Solving Techniques for CSP Refinement Checking

3.1 Introduction

To alleviate the state-space explosion problem inherent in model checking, a significant number of techniques have been proposed. Methods for decreasing the size of the generated state space and enhancing the model checking algorithm include CEGAR [CGJ$^+$00], partial-order reductions [CGP99, Pel98], bounded model checking [BCCZ99], etc. Regarding state-space representation, the major dichotomy is between explicit and symbolic [BCM$^+$92, BCCZ99] model checking. *Explicit model checking* is based on explicit enumeration and examination of individual states. *Symbolic model checking* relies on abstract representation of sets of states and sets of transitions, generally as Boolean formulas, and properties are validated using techniques such as BDD manipulation or SAT solving. In many practical applications, the symbolic representation of a system is exponentially more succinct than the explicit one [CGJ$^+$03].

The implicit state-space representation was initially based on BDDs—graph-based data structures providing canonical representation of Boolean functions, given a fixed variable ordering [Bry86]. BDD-based model checking has proven to be extremely powerful in verifying synchronous hardware systems [BCM$^+$92]. BDDs offer the advantage of yielding compact representation and allowing efficient manipulation, also capturing regularities in the system [CGP99, Par02]. A major drawback of this approach is that the size of the BDD is dramatically influenced by the ordering of the Boolean variables. Model checkers employing BDDs rely on heuristics (or dynamic reordering strategies) for finding variable orderings that provide compact state-space representation. However, in general, finding an optimal variable ordering for a Boolean function is NP-complete. For many examples no space-efficient variable orderings exist [BCCZ99].

The recent advances of efficient SAT solvers have significantly broadened the horizons of symbolic model checking. The problem of Boolean satisfiability is in general NP-complete [Coo71]; however, modern SAT solvers employ various heuristics and perform exceptionally well in practice, both for satisfiable and unsatisfiable instances. SAT-based bounded model checking (BMC) [BCCZ99] has proven to be an extremely powerful technique, mainly suited, due to its incompleteness, to falsification of properties. Approaches for making BMC complete include calculating completeness thresholds [CKOS04, CKOS05] or augmenting BMC with k-induction [SSS00, ES03b] or Craig interpolation [McM03] techniques. Both k-induction and interpolation-based model checking build upon BMC, aim at establishing inductiveness of properties and are capable of both bug finding and establishing the correctness of systems.

Both bounded and unbounded SAT-based model checking have been mainly investigated in the context of hardware and sequential software systems.

NuSMV [CCGR00, CCG$^+$02] is a symbolic model checker for analysing systems defined in the SMV input language. SMV essentially allows for modelling deterministic and nondeterministic transition systems and composing those in a modular and hierarchical fashion, also supporting both synchronous and asynchronous form of concurrency through shared variables. In its general framework, NuSMV verifies systems against CTL properties using BDD-based model checking. The SAT-based framework supports BMC and k-induction and uses specifications written in LTL. Despite the support for concurrency, the latter two techniques have been mostly evaluated in the context of hardware circuits.

Cbmc [CKY03, CKL04] is a bounded model checker that verifies ANSI-C and C++ programs against user-specified assertions, as well as for general correctness properties such as pointer safety, array indices in bounds, etc. Cbmc supports all ANSI-C operators and pointer constructs, including dynamic memory allocation and pointer arithmetic; however the tool currently does not offer support for handling multi-threaded programs. A version of Cbmc for reasoning about concurrent C programs operating within a bounded number of context switches was implemented by IBM in a tool called Tcbmc [RG05], however it was restricted only to the analysis of programs with two threads. Another extension of Cbmc, implemented in a tool called K-Inductor [DHKR11], features the k-induction technique for C and C++ programs, but with no support for concurrency. The work of [DHKR11] also presents K-Boogie, an extension of the Boogie verifier [BCD$^+$05] with support for k-induction; again, this is in the context of sequential programs.

Temporal k-induction for a limited form of concurrency has been investigated in the setting of establishing DMA-race freedom in multi-core programming, where cores communicate and synchronise via shared memory and barrier primitives [DKR10, DKR11a]. In this setting there is concurrency due to multiple cores and due to multiple DMA operations that are simultaneously pending. The work of [DKR10, DKR11a], and the resulting tool scratch [DKR11b], are restricted to analysis of a *single* thread that issues concurrent DMA operations. This limited form of concurrency is captured via an encoding into a sequential program.

We address the problem of applying BMC and temporal k-induction [ES03b] to concurrent systems in the process-algebraic setting of CSP and the state-of-the-art refinement checker FDR, with focus on the traces model of CSP, which is sufficient for verifying safety properties. To the best of our knowledge, this is the first application and experimental evaluation of k-induction in the setting of concurrent software systems.

Regarding the world of CSP, to the best of our knowledge, our implementation of the k-induction algorithm is the first attempt of applying unbounded SAT-based refinement checking to CSP. In general, the core of FDR is refinement checking in each of the semantic models, which is carried out on the level of the operational representation of the CSP processes and is implemented using explicit state enumeration supplemented by hierarchical state-space compression techniques. Although until now FDR has followed the explicit model checking approach, there has been some work on the symbolic model checking of CSP resulting in the BDD-based refinement checker ARC [PY96] and the model checker PAT [SLDS08]. To the best of our knowledge, PAT is the only tool that applies BMC to compositional process algebras with recursion and hiding (but not renaming). In general PAT verifies systems defined in a version of CSP enhanced with shared variables. Within the BMC framework, PAT uses specifications defined as reachability properties on the values of the shared variables, which requires a different model checking algorithm based on reachability and not on language containment.

Traditionally, refinement checking in CSP reduces to checking for reverse containment of possible behaviours. Hence, we need to exploit the SAT solver to decide bounded language inclusion as opposed to bounded reachability of error states, as in most existing model checkers. Due to the harder problem to decide and the presence of invisible silent actions in process algebras, the original syntactic translation of BMC to SAT cannot be applied directly and we adopt a semantic translation algorithm based on watchdog transformations [RGM$^+$03]. Essentially, this involves reducing a refinement check into analysing a single process which is constructed by putting the implementation process in parallel with a transformed specification process. The latter plays the role of a watchdog that monitors and marks violating behaviours.

In terms of Boolean encoding of CSP processes, both PAT and ARC exploit a fully compositional encoding of CSP processes. We propose a new encoding scheme resting on FDR's hybrid two-level approach for calculating the operational semantics using supercombinators.

We have implemented BMC and temporal k-induction in a tool called SymFDR which builds upon FDR to obtain an alternative symbolic refinement engine. The symbolic engine [POR09, POR12] mimics FDR up to the state-space exploration phase and adopts FDR's implicit operational representation based on supercombinators [Ros11b]. For both BMC and k-induction, SymFDR offers configurable support for a SAT solver (MiniSAT 2.0 [ES03a, EB05], PicoSAT 846 [Bie08] or ZChaff [MMZ$^+$01], all used in incremental mode), Boolean encoding (one-hot or binary), traversal mode (forward or backward), etc.

Experiments indicate that the BMC engine sometimes substantially outperforms the original explicit state-space exploration method adopted by FDR, especially for complex tightly-coupled combinatorial problems, as reported in [POR09, POR12]. For k-induction, the completeness threshold blows up in all cases, due to concurrency, and, therefore, high performance depends on whether or not the property is k-inductive for some small value of k. Hence, the SAT engine generally scales better only when a counterexample exists. We compare the performance of SymFDR with the performance of FDR, FDR used in a non-standard way, PAT [SLD08] and, in some cases, NuSMV [CCG$^+$02], Alloy Analyzer [Jac06] and straight SAT encodings tailored to the specific problems under consideration.

3.2 Foundations

3.2.1 SAT Essentials

SAT Solvers. A SAT solver is a decision procedure that takes as input a Boolean formula φ and decides whether φ is satisfiable or not. In the affirmative, the SAT solver reports a satisfying assignment of φ's variables. In the negative, some SAT solvers are also able to output an unsatisfiable core of clauses or generate a resolution proof of unsatisfiability for φ. The problem of Boolean satisfiability is in general NP-complete [Coo71]; however, modern SAT solvers employ various heuristics and perform exceptionally well in practice, both for satisfiable and unsatisfiable instances. The vast majority of SAT solvers require that Boolean formulas be input in conjunctive normal form (CNF). Methods for transforming a Boolean formula to an equivalent or equisatisfiable one in CNF are presented in Section 2.1.4.

The DPLL Algorithm. Most modern SAT solvers are based on the DPLL framework [DP60, DLL62]. DPLL solvers essentially operate by making a decision on a variable, propagating the implications of this decision by repeatedly applying the unit clause rule, and backtracking if reaching a conflict. The unit clause rule is the only inference rule employed and its repeated application is also known as Boolean constraint propagation (BCP). Upon encountering a conflict, the SAT solver uses binary resolution to analyse and identify the reason for the conflict, generate a conflict clause and add it to the SAT instance, the process of which is called *learning*. Learnt clauses are implied by the original SAT instance, hence their addition is sound. Moreover, their explicit recording in the clause database makes the search for a satisfying assignment more deterministic, i.e., prunes the search space. After learning the reason for the conflict, the SAT solver backtracks by progressively invalidating decisions on variables up to a certain decision level and restarts the process.

Different SAT solvers exploit different heuristics for choosing the variables to decide, the order of clauses to resolve during BCP, the conflict clauses to learn, the decision level to return to when backtracking, etc. The SAT solver MiniSAT 2.0 [ES03a, ES03b], for example, is strongly conflict-driven in all its choices of heuristics. In general, the efficiency of recent DPLL SAT solvers is to a large extent a result of their capabilities to learn from conflicts, decide on important variables, prune large pieces of the search-space, and restart the solver in order to escape from parts of the search-space that do not seem promising. The interested reader is referred to [KS08, BHvMW09] for more details.

Incremental SAT Solving. Incremental SAT solving can be employed whenever there is a need to solve a series of related SAT instances. Current SAT instances are then defined on top of previous ones. The technique allows for reusing important information such as learnt conflict clauses and statistical data used in heuristics, which might be beneficial not only for solving the current SAT instance, but also for solving subsequent similar SAT instances. Maintaining information deduced from previous runs might prevent the SAT solver from taking the same fruitless decisions and traversing the same search space all over again.

An illustration of incremental SAT solving is the algorithm for obtaining all satisfying assignments of a Boolean formula. The SAT solver is run iteratively until the SAT instance becomes unsatisfiable. Every time the SAT solver reports a satisfying assignment, a clause prohibiting this assignment is added to the clause database and the solver is rerun.

MiniSAT provides an additional incremental SAT-solving interface. A set of literals can be passed as an argument to the solving procedure. Those literals are called *assumptions* and can be viewed as unit clauses. The solver assumes that the literals are true throughout the duration of the SAT check. After the check is over, the assumptions are withdrawn automatically, even if the formula is unsatisfiable under those assumptions. The advantage of this mechanism is that clauses learnt during the SAT check can also safely be preserved. The reader is referred to [ES03a] and [ES03b] for more information.

3.2.2 Bounded Model Checking

Bounded model checking (BMC) is a sound but generally incomplete technique that focuses on searching for counterexamples of bounded length only. The underlying idea is to fix a bound k and unwind the implementation model for k steps, thus considering behaviours and counterexamples of length at most k. In practice, BMC is conducted iteratively by progressively increasing k until one of the following happens: (1) a counterexample is detected, (2) k reaches a precomputed threshold called *completeness threshold* [CKOS04, CKOS05], which indicates that the model satisfies the specification, or (3) the model checking instance becomes intractable.

Given an implementation model and a property to verify, a *completeness threshold* [CKOS04, CKOS05] is any natural number $\mathcal{CT}$ that guarantees that, if there is no violation of the property of length $\mathcal{CT}$ or less, then there is no violation of greater length. Different notions of completeness threshold exist, mainly based on the properties of the underlying graph of the system, e.g., *diameter* (the longest shortest path between any two states), *recurrence diameter* (the longest simple path between any two states), forward and backward radius versions of both, the size of the state space, etc. [BCCZ99, CKOS04, CKOS05, BHvMW09]. The *forward radius* is the longest shortest path to any state starting from an initial one and the *backward radius* is the longest shortest path from any state to a state violating the property in question. A *simple path* is a path along which all states are different and, in general, the recurrence diameter of a graph can be arbitrarily longer than its diameter (the same holding for radii)—if we consider a clique of size n, it's diameter would be 1, while the recurrence diameter would be $(n-1)$. We remark that this problem is exacerbated when modelling concurrent systems due to the exponential blow up of the state space.

It is important to note that without knowing or reaching a completeness threshold, the BMC procedure is incomplete since we do not know at what step it is correct to stop iterating and declare that the system preserves the desired property. Therefore, BMC is mostly suitable for detecting bugs rather than for full verification, i.e., proving the absence of bugs.

The problem with completeness thresholds is two-fold. On one hand, calculating the exact completeness threshold can be as hard as the model checking problem itself [CKOS05] and, therefore, sound overapproximations of it are usually used in practice. On the other hand, in some cases those overapproximations can be too large to handle efficiently.

SAT-based BMC [BCCZ99] reduces the model checking problem to a propositional satisfiability problem. At each step k, a Boolean formula is constructed which models a counterexample of length k. This formula is fed into a SAT solver which decides the model checking problem in question and instantiates a counterexample, if any. Due to the DFS-nature of the SAT decision procedure, this technique allows for fast detection of counterexamples. Moreover, due to the iterative nature of the BMC framework, the counterexample generated is of minimal length.

In the original syntactic [BCCZ99] and the subsequent semantic [CKOS04, CKOS05] translation of BMC to SAT, the implementation is modelled by a Kripke structure M and verified against a specification f defined as an LTL formula. The BMC instance at each step k is translated to a Boolean formula $\varphi_k = [\![M]\!]_k \wedge [\![\neg f]\!]_k$, where $[\![M]\!]_k$ encodes all paths of M of length k and $[\![\neg f]\!]_k$ represents all paths of length up to k that violate f. Hence, φ_k is satisfiable if and only if there is a path of of M of length k that violates f.

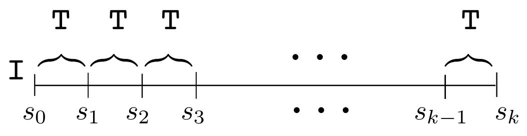


Figure 3.1: Encoding paths of length k

Generally, having a Boolean encoding of the state space (e.g., a binary or a one-hot encoding [KB05]), the Kripke structure M can be represented symbolically by a pair of Boolean functions $\langle I(s), T(s, s')\rangle$ defined as the characteristic functions of the set of initial states and the transition relation, respectively. We use s and s' as a shorthand for the vectors of Boolean variables necessary for encoding states of M. We replicate a separate copy of state variables s_i for each time step i. Then $[\![M]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$ (see Figure 3.1). We illustrate the structure of the entire formula φ_k with a simple example in case $f = \mathsf{G}p$, where p represents a state predicate with Boolean encoding P:

$$\varphi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg P(s_i) \tag{3.1}$$

3.2.3 From Bounded to Complete Model Checking

3.2.3.1 Temporal k-induction

Temporal k-induction [SSS00, ES03b] is a complete SAT-based technique for verifying safety properties. As opposed to BMC, it can be used also for establishing correctness of systems. Given a model $\langle I(s), T(s, s')\rangle$ and a safety property $P(s)$, the method checks if all reachable states of the model preserve P. k-induction builds upon BMC and is also conducted iteratively, as presented in Algorithm 4. It provides two conditions for termination in case the property is not violated—k-inductiveness of P for some $k \in \mathbb{N}$ or reaching the backward recurrence radius of the model with respect to P. The property P is *k-inductive* if it can be proven that if P holds along all initial paths of the system of length k, then it cannot be violated on a path of length $k+1$. The *backward recurrence radius* is the length of the longest simple path from any state to a state violating P and is a valid completeness threshold. It is dependent on both the model of the system and the property under consideration.

Algorithm 4 Temporal k-induction [ES03b]

```
1: for $k = 0$ to $\infty$ do
2:    if $satisfiable(Base_k)$ then
3:       return PROPERTY VIOLATED and a counterexample trace
4:    end if
5:    if $unsatisfiable(Step_k \wedge Simple_k)$ then
6:       return PROPERTY HOLDS
7:    end if
8: end for
```

For each step k, the temporal induction proof consists of two parts—a base case and an induction step. The base case $Base_k$ is similar to a BMC instance—we check if, starting from an initial state, there is a path of length k that violates P (see Figure 3.2). In the base case, we assume that we have already checked all base cases of shorter length and strengthen the BMC instance by stating that P holds along all initial paths of length up to $k-1$. If the base case is satisfiable, we have found a counterexample. Otherwise, we proceed with the induction step $Step_k$ which is designed to prove that P is k-inductive. The induction step is strengthened and made complete by a constraint $Simple_k$ requiring that all states on the $(k+1)$-path be different. Hence, k-induction terminates with a positive answer when reaching the backward recurrence r radius even if the property P has not manifested itself as k-inductive for any $k \leq r$.

$$\begin{array}{lcl}
Base_k & \widehat{=} & I(s_0) \;\wedge\; \left(\bigwedge_{0\leq i<k} \; (P(s_i) \wedge T(s_i, s_{i+1}))\right) \;\wedge\; \neg P(s_k) \\
Step_k & \widehat{=} & \qquad\quad \left(\bigwedge_{0\leq i<k+1} (P(s_i) \wedge T(s_i, s_{i+1}))\right) \;\wedge\; \neg P(s_{k+1}) \\
Simple_k & \widehat{=} & \bigwedge_{0\leq i<j\leq k} (s_i \neq s_j)
\end{array}$$

Figure 3.2: k-induction ingredients [ES03b]

We remark again that, in many cases, the *backward radius* of the model—the longest shortest path from any state to a state violating P, can be considerably smaller than its backward recurrence radius. However, the translation of shortest paths between two states to SAT involves plenty of existential quantifiers and is mostly suitable to using a QBF engine instead of a SAT solver.

As we are dealing with safety properties, we can also carry out the k-induction algorithm backwards, starting from states that violate P and trying to prove that initial states are never (backward) reachable. This can be implemented by redefining $Base_k$ and $Step_k$ as depicted in Figure 3.3. This algorithm guarantees termination upon reaching the *forward recurrence radius*—the longest simple path to any state starting from an initial state.

$$\begin{array}{lcl}
Base_k & \widehat{=} & \neg P(s_0) \;\wedge\; \left(\bigwedge_{0\le i<k} \;(\neg I(s_i) \wedge T^{-1}(s_i, s_{i+1}))\right) \;\wedge\; I(s_k) \\
Step_k & \widehat{=} & \left(\bigwedge_{0\le i<k+1} (\neg I(s_i) \wedge T^{-1}(s_i, s_{i+1}))\right) \;\wedge\; I(s_{k+1}) \\
Simple_k & \widehat{=} & \bigwedge_{0\le i<j\le k} (s_i \neq s_j)
\end{array}$$

Figure 3.3: Backward k-induction ingredients

3.3 Bounded Trace Refinement Framework

In this section, we present our iterative bounded refinement checking algorithm. Our approach for establishing trace refinement is based on watchdog transformations [RGM+03]. Our objective is the following. We are given two CSP processes *Spec* and *Impl* and an integer k. We aim at checking whether $Spec \sqsubseteq_T^k Impl$, i.e., whether all executions of the implementation of length at most k agree with the specification. Similarly to BMC and k-induction, we carry out the analysis on the level of the operational representation of *Spec* and *Impl*. We point out that executions of length k can correspond to traces of smaller length if having τ actions entangled within, as defined in Section 2.2.4.2.

3.3.1 Challenges

As the LTS's underlying the operational semantics of processes are event-based models, we need to also handle events in our encoding. Let $OS_{Spec} = \langle I^s(s), T^s(s, l, s')\rangle$ and $OS_{Impl} = \langle I^i(t), T^i(t, l, t')\rangle$ be the models of *Spec* and *Impl*, respectively. At first glance, the most natural approach for encoding bounded execution refinement would be to try and directly mirror the original translation of BMC to SAT. We would need to similarly construct the Boolean formula φ_k as a conjunction of two formulas to model all executions of *Impl* of length k that are not executions of *Spec*:

$$\varphi_k = [\![OS_{Impl}]\!]_k \wedge [\![\neg OS_{Spec}]\!]_k.$$

Hence, we would be looking for an instantiation of the vectors of Boolean variables $l_1, \ldots, l_k$, such that $[\![OS_{Impl}]\!]_k = I^i(t_0) \wedge \bigwedge_{i=0}^{k-1} T^i(t_i, l_i, t_{i+1})$ is satisfiable and $[\![OS_{Spec}]\!]_k = I^s(s_0) \wedge \bigwedge_{i=0}^{k-1} T^s(s_i, l_i, s_{i+1})$ is not. Due to the implicit universal quantification of $s_0, \ldots, s_k$ in the unsatisfiability check of $[\![OS_{Spec}]\!]_k$, this analysis is mostly suitable to a QBF engine. Using a SAT solver in this case would mean that we would need to extinguish all possible satisfying assignment of $l_1, \ldots, l_k$ in $[\![OS_{Impl}]\!]_k$ and prove the unsatisfiability of $[\![OS_{Spec}]\!]_k$ over each one of them.

Furthermore, invisible τ actions can be arbitrarily interleaved in the executions of *Impl* and, therefore, syntactically different executions can produce semantically equivalent traces. This can lead to reporting spurious counterexamples. To illustrate this, consider the executions $\langle a, \tau, \tau, b, \tau, c\rangle$ of *Impl* and $\langle a, b, c\rangle$ of *Spec*. Even though they correspond to the same trace $\langle a, b, c\rangle$, they do not match pointwise and $\langle a, \tau, \tau, b, \tau, c\rangle$ would be falsely reported as a violation of *Spec*. However, bookkeeping the possible sequences of τ-s stuttered in between visible events does not seem to be trivial and computationally justifiable on the level of Boolean functions.

3.3.2 Watchdog Refinement-Checking Algorithm

As explained in Section 2.3.2, FDR performs the refinement check by normalising the specification and looking for the existence of behaviours that the implementation allows and the specification does not.

As an alternative, the watchdog approach [RGM$^+$03, Ros11b] reduces the refinement check to analysing a single process constructed by composing the implementation in parallel with a transformed specification process. The latter plays the role of a watchdog that monitors the implementation and flags any behaviours that are considered violating with respect to the specification.

In our settings, using watchdog transformations allows us to actually reduce bounded execution containment to bounded reachability which is already amenable to SAT. The watchdog transformation phase is performed by means of FDR.

3.3.2.1 Preprocessing Phase Using FDR

Our implementation is intended as an alternative back-end for FDR, orthogonal to the standard explicit method of performing trace refinement. Currently, we use a shared library version of FDR for manipulating CSP processes and we mimic FDR up to the point of the final state-space exploration phase. Therefore, SymFDR reuses FDR's compiler and supercompiler and the data structures underlying the hybrid two-level operational representation of processes, consisting of a process tree and a set of supercombinators, as defined in Section 2.3.1.2.

At present, we use FDR to supercompile and normalise *Spec* and to retrieve OS_{Spec} representing the operational semantics of *Spec*.

Without loss of generality, we assume that the implementation *Impl* comprises the interaction of c sequential processes $P_1, \ldots, P_c$ running in parallel, possibly using hiding, renaming or other CSP operators other than recursion. We write $Impl = P_1 \parallel P_2 \parallel \ldots \parallel P_c$ to actually denote a high-level process *Impl* with leaf processes $P_1, \ldots, P_c$, as defined in Section 2.3.1.2. This form of representing concurrent systems after supercompilation is of no limitation—we can handle the entire CSP$_M$ syntax and functionality apart from the function *chase*. We use FDR to supercompile *Impl* and to retrieve both the set of supercombinators and the set $\{OS_{P_i} \mid i \in \{1, \ldots, c\}\}$.

3.3.2.2 Watchdog Bounded Refinement-Checking Algorithm

In a nutshell, the main steps of our algorithm are the following:

1. We transform *Spec* into a process *Watchdog* which allows the behaviours of both *Spec* and *Impl* and, in fact, many others, but marks those that do not conform to *Spec*. The transformation is carried out on the level of the LTS and not on the higher CSP description of *Spec*. It is most easily defined if the specification process is normalised so that it reaches a unique state after following any trace. The LTS of *Watchdog* is then obtained as an extension of OS_{Spec}—we add a fresh state *sink* and make OS_{Spec} total with respect to the alphabet $\alpha_{Spec} \cup \alpha_{Impl}$ by directing all non-existing transitions to *sink*. Formally, having $OS_{Spec} = \langle S^s, s_0^s, T^s, L^s = \alpha_{Spec} \rangle$,

$$OS_{Watchdog} = \langle S^w = S^s \cup \{sink\}, s_0^s, T^w, L^w = \alpha_{Spec} \cup \alpha_{Impl} \cup \{\tau\} \rangle,$$

where the transition relation T^w is defined as follows:

$$\begin{aligned} T^w &= T^s \cup \{(\mathit{sink}, l, \mathit{sink}) \mid l \in \alpha_{\mathit{Spec}} \cup \alpha_{\mathit{Impl}} \cup \{\tau\}\} \cup \\ &\quad \{(s, l, \mathit{sink}) \mid s \in S^s, l \in \alpha_{\mathit{Spec}} \cup \alpha_{\mathit{Impl}}, s \overset{l}{\nrightarrow}\} \end{aligned}$$

The resulting process *Watchdog* exhibits all traces over the alphabet $\alpha_{\mathit{Spec}} \cup \alpha_{\mathit{Impl}}$. However, since we start with a normalised OS_{Spec}, the resulting process *Watchdog* operationally passes through *sink* whenever executing a trace that is not allowed by *Spec*.

2. We construct a process $\mathit{Refinement} = \mathit{Watchdog} \underset{\alpha_{\mathit{Impl}} \cup \alpha_{\mathit{Spec}}}{\|} \mathit{Impl} = \mathit{Watchdog} \underset{\alpha_{\mathit{Impl}} \cup \alpha_{\mathit{Spec}}}{\|} (P_1 \parallel P_2 \ldots \parallel P_c)$. *Refinement* captures precisely the traces of *Impl*, but those traces that do not conform to *Spec* force *Watchdog* to bark, i.e., pass through its *sink* state. Hence, *Refinement* can be used as an indicator whether *Impl* can behave in a way incompatible with *Spec*. *Watchdog* becomes just one of the sequential leaf processes of *Refinement*. It is evident then that:
   (a) $\mathit{Spec} \sqsubseteq_T \mathit{Impl} \iff$ *Watchdog* never reaches its *sink* state in any execution of *Refinement*.
   (b) All executions of *Refinement* forcing *Watchdog* to pass through its *sink* state constitute valid counterexamples of the assertion $\mathit{Spec} \sqsubseteq_T \mathit{Impl}$.
   (c) The process *Refinement* can never deadlock on an erroneous trace, i.e., on a trace that passes through *sink* (this follows from the fact that we allow an execution of *Watchdog* to contain any number of τ's after visiting *sink*).

3. We check whether *Watchdog* can reach its *sink* state within k steps of the execution of *Refinement*. Because *Refinement* cannot deadlock once it has passed through *sink*, we can unwind its transition relation a configurable number i steps at once and then use the SAT solver to check whether any of the last i transitions signals for an error. In contrast, in the original version of BMC, the SAT solver is invoked upon every subsequent unfolding of the transition relation.

The subtle point and the danger here is to avoid reporting false negatives due to excessive unfolding of the transition relation. Let us recall (see Section 3.2.2) that the reduction of BMC to SAT is founded upon generating counterexamples from satisfiable SAT instances. More in particular, having a safety property requiring that P never happens, the original version of BMC models a possible counterexample of length up to k as follows:

$$\varphi_k = I(s_0) \;\wedge\; \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \;\wedge\; \bigvee_{i=0}^{k} \neg P(s_i).$$

Hence, φ_k is unsatisfiable precisely whenever there are no counterexamples of length up to k. Now suppose that we have a safety property of this form that fails due to a single counterexample path of length 7 and all paths of the system under consideration are of length at most 8. Let us further suppose that with the original BMC algorithm we unfold 5 steps at once and only then do we check whether any of the states reachable in the last 5 steps is erroneous. Then, φ_5 would be unsatisfiable, implying, soundly, that there are no errors of length 5 or less. However, φ_{10} would be unsatisfiable as well because the subformula $I(s_0) \;\wedge\; \bigwedge_{i=0}^{10} T(s_i, s_{i+1})$ would be itself unsatisfiable as all paths of the system are of length 8 or less. This would imply, this time unsoundly, that there are no errors of length 10 or less. Hence, the erroneous path of length 7 would never be reported and the algorithm would, spuriously, mark the system as correct (upon hitting a completeness threshold).

In contrast, our watchdog construction guarantees that for any $k \in \mathbb{N}$, paths of length k of the process *Refinement* always exist. Hence the formula $I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$, tailored to *Refinement*, is always satisfiable and an error can never be missed, even if unfolding multiple transitions at once. In the general case, a counterexample generated out of *Refinement* is just padded with τ's in the end, if not of the necessary length. As mentioned above, this follows from the fact that we allow an execution of *Watchdog* to contain any number of τ's after visiting *sink*, as well as from the fact that τ, as an invisible action, never requires synchronisation.

3.4 Boolean Encoding of CSP Processes

In this section we present our translation of CSP processes into Boolean formulas. The Boolean encoding can be used for both SAT-based and BDD-based model checking. First, we demonstrate how to encode sequential processes, corresponding to leaf processes in the operational representation. Then, we show how to glue together sequential processes with supercombinators to obtain an encoding of a high-level process. In what follows, we call a high-level process a concurrent system.

Let us first introduce the necessary notation. We will write $X(\mathit{Vars})$ to denote the Boolean encoding of X with respect to the vector(s) of Boolean variables Vars. To simplify the notation, given a vector of Boolean variables $\overline{x} = (x_1, \ldots, x_m)$, we will write x to denote $\overline{x}$. Given a Boolean formula f over x and a bit vector $b \in \{0,1\}^m$, we will write $f(x)|_{x=b}$ to denote the value of $f(b)$. For a given $n \in \mathbb{N}$, we will assume that $\lceil \log_2 n \rceil = 1$ if $n = 1$.

3.4.1 Encoding Sets

Let S be a finite set of elements. In this section we recall standard techniques for representing elements and subsets of S as characteristic Boolean functions.

In order to obtain a Boolean encoding of S, we first employ an injective function $\mathit{enc}_s : S \mapsto \{0,1\}^m$ that maps each element of S into a unique bit vector $b = (b_1, \ldots, b_m)$ of size $m \in \mathbb{N}$. Then we introduce an ordered vector $x = (x_1, \ldots, x_m)$ of m distinct Boolean variables, where each variable x_i serves for identifying the corresponding i-th bit of a bit vector $b \in \mathit{enc}_S(S)$. Given an element $s \in S$, its Boolean encoding $s(x)$ satisfies the following: $s(x)|_{x=b} =$ true if and only if $\mathit{enc}_S(s) = b$.

Typically binary or one-hot [KB05] encoding of sets are most commonly used in practice.

The basic idea behind *binary encoding* is to enumerate the elements of S in binary notation and represent them as Boolean functions over $m = \lceil \log_2 |\, S \,| \rceil$ Boolean variables. As an example, let us consider the set $S = \{s_0, s_1, s_2, s_3\}$ and the binary encoding function enc mapping s_0 to (00), s_1 to (01), s_2 to (10) and s_3 to (11). In order to encode elements of S as functions, a vector of just two variables $x = (x_0, x_1)$ suffices and $s_1(x) = \neg x_0 \wedge x_1$, for example. A subset S' of S is encoded by taking the disjunction of the encodings of the elements in S'. For example, $\{s_0, s_1\}(x) = \neg x_0$.

In *one-hot encoding*, each element $s \in S$ is represented by a bit vector of size $|\, S \,|$ in which precisely one bit is set to 1. To illustrate the technique, let us take the same set $S = \{s_0, s_1, s_2, s_3\}$ and a one-hot encoding function enc mapping s_0 to (1000), s_1 to (0100), s_2 to (0010) and s_3 to (0001). In this case we will need a vector $x = (x_0, x_1, x_2, x_3)$ of four variables and $s_1(x) = \neg x_0 \wedge x_1 \wedge \neg x_2 \wedge \neg x_3$. Alternatively, we can use $s_1(x) = x_1$, but in this case we need to add global constraints enforcing that precisely one bit is set to true at a given time instance. Those constraints can be expressed by a formula of size linear in $|S|$ [BHvMW09]. In contrast to binary encoding where $\neg x_0$ encodes the set of elements $\{s_0, s_1\}$, with one-hot encoding $\neg x_0$ encodes the *set of sets* of elements $\{S' \subseteq S \mid s_0 \notin S'\}$ and $\{s_0, s_1\}(x) = x_0 \wedge x_1 \wedge \neg x_2 \wedge \neg x_3$. In general, let us note that binary encoding is limited to encoding sets of elements and one-hot encoding—to sets of sets of elements.

3.4.2 Encoding a Sequential Process

As described in Section 3.3.2.1, for each sequential leaf process P, we obtain the explicit operational representation of P using FDR. Let $OS_P = \langle S, \mathsf{init}, L = \Sigma^{\tau,\checkmark}, T\rangle$ be the LTS associated with the finite-state leaf process P communicating over a finite alphabet of events Σ. Using either binary or one-hot encoding of sets, we introduce vectors of Boolean variables x and y for encoding the set of states S and the set of labels L, respectively. We define $I(x) = \mathsf{init}(x)$.

In order to represent the transition relation T, we employ a copy x' of x. x serves to represent the source states of transitions and x' the destination states. Then, given a transition $t = (s_{src}, l, s_{dest}) \in T$,

$$t(x, y, x') = s_{src}(x) \wedge l(y) \wedge s_{dest}(x').$$

For any $s \in S$, we write $s(x')$ to denote $s(x)[x' \leftarrow x]$, i.e., we represent s with respect to the variables x and then substitute the variables x with x'. The encoding of the entire transition relation is the following:

$$T(x, y, x') = \bigvee_{t \in T} t(x, y, x').$$

We can now represent a sequential process P implicitly by the pair of Boolean functions $\langle T^P(x, y, x'), I^P(x)\rangle$. Given $k \in \mathbb{N}$, we define $\mathit{Paths}(P, k)$ to be the set of all executions $\mathsf{init} = s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \ldots \xrightarrow{l_k} s_k$ of OS_P of length k. Let us replicate $(k+1)$ vectors of Boolean variables $x_0, x_1, \ldots x_k$ for encoding the states $s_0, s_1, \ldots, s_k$ and k vectors of Boolean variables $y_1, y_2 \ldots y_k$ for encoding the corresponding transitions $l_1, \ldots, l_k$. Then the symbolic representation of $\mathit{Paths}(P, k)$ is the following:

$$\mathit{Paths}(P, k)(x_0, x_1, \ldots, x_k, y_1, y_2, \ldots, y_k) = I^P(x_0) \wedge \bigwedge_{i=0}^{k-1} T^P(x_i, y_{i+1}, x_{i+1}).$$

3.4.3 Encoding a Concurrent System

In the setting of FDR, after supercompilation we can view a concurrent system as a high-level process identified by a process tree and a set of supercombinators. Since a high-level root process can be modelled as an LTS, we now show how to encode a concurrent system similarly to a low-level sequential process. In what follows, we denote by $System(c) = \langle\langle P_1, \ldots, P_c\rangle, SC\rangle$ the high-level process characterised by a set of supercombinator rules SC and c explicitly compiled leaf processes $P_1, \ldots, P_c$ communicating over sets of events $\Sigma_1, \ldots, \Sigma_c$, respectively. We define $\Sigma = \bigcup_{i=1}^{c} \Sigma_i$.

Encoding the Sequential Leaf Processes. For each $i \in \{1, \ldots, c\}$, we retrieve the explicit LTS representation $OS^i = \langle S^i, \mathsf{init}^i, L^i = \Sigma_i^{\tau,\checkmark}, T^i\rangle$ of the leaf P_i from FDR. Since $\Sigma_i \subseteq \Sigma$, we actually consider $L^i = \Sigma^{\tau,\checkmark}$.

Following the ideas from Section 3.4.2, we introduce vectors of Boolean variables x^i, $x^{i'}$ and y^i to generate the symbolic representation $\langle T^i(x^i, y^i, x^{i'}), I^i(x^i)\rangle$ of P_i. Hence, each process has its own set of variables for representing the alphabet $\Sigma^{\tau,\checkmark}$. We further introduce an additional vector of Boolean variables y for encoding the resulting action of the entire system because, due to the presence of hiding and renaming, it might be different from the contributions of the leaf processes, as illustrated in Section 2.3.1.2. In case the system violates the specification, we generate a counterexample trace out of the satisfying assignment of the variables from y.

Encoding Configurations of the Concurrent System. Recall that at, every time instance, the state of the entire high-level system, also called a *configuration*, is identified by the current states of its sequential leaf components. Formally, the set of states of the system is a c-ary relation $S \subseteq S^1 \times \ldots \times S^c$, the initial state being $\mathsf{init} = (\mathsf{init}^1, \ldots, \mathsf{init}^c)$. Therefore, S can be represented symbolically using the Boolean variables from $x^1, \ldots, x^c$. If $s = (s^1, \ldots, s^c) \in S$, then $s(x^1, \ldots, x^c) = \bigwedge_{i=1}^{c} s^i(x^i)$. For clarity, we denote the set of states of the overall system by $Cfgs$.

Supercombinators and Formats. As we mentioned in Section 2.3.1.2, supercombinators are rules for combining together actions of the individual sequential leaf processes into event-outcomes of the overall system [Ros98]. Within a supercombinator, each process can participate with a visible event, a silent action τ, or not be involved at all. We denote the non-involvement with the symbol ϵ. For any alphabet Σ, we let $\Sigma^\epsilon = \Sigma \cup \{\epsilon\}$. In addition, the set of supercombinators is partitioned into existing *formats*, i.e., different configurations of switched-on and switched-off processes among $P_1, \ldots, P_c$, which we denote by F.

Formally, the set of supercombinators can be represented as a $(c+3)$-ary relation SC:

$$SC \subseteq F \times \Sigma_1^{\tau,\checkmark,\epsilon} \times \ldots \times \Sigma_c^{\tau,\checkmark,\epsilon} \times \Sigma^{\tau,\checkmark} \times F,$$

or more generally:

$$SC \subseteq F \times (\Sigma^{\tau,\checkmark,\epsilon})^c \times \Sigma^{\tau,\checkmark} \times F.$$

$(f_{src}, a_1, \ldots, a_c, a, f_{dest}) \in SC$ if and only if from a certain configuration and a certain format f_{src} of the overall system, P_1 performs a_1, $\ldots$, P_c performs a_c and the overall system performs a switching to a format f_{dest}. The set of supercombinators for Milner's scheduler was illustrated earlier in Example 2.3.2.

The operational semantics of the concurrent system can be considered an implicit LTS, whose transitions can be switched on and off:

- set of states: $F \times \mathit{Cfgs}$
- set of labels: SC
- transition relation: $T \subseteq (F \times \mathit{Cfgs}) \times SC \times (F \times \mathit{Cfgs})$. If the system is in a given configuration and in a given format, the individual processes' transition relations determine if the labels are switched on or off. Formally,

$$(f_i, (s_i^1, \ldots, s_i^c)) \xrightarrow{(f_i, a_1, \ldots, a_c, a, f_j)} (f_j, (s_j^1, \ldots, s_j^c))$$

iff

$$(f_i, a_1, \ldots, a_c, a, f_j) \in SC \wedge \forall_{k=1}^{c}((a_k \neq \epsilon \Rightarrow (s_i^k, a_k, s_j^k) \in T^k)\ \wedge (a_k = \epsilon \Rightarrow s_i^k = s_j^k)).$$

To illustrate the concept, a transition of Milner's scheduler was modelled as Example 2.3.3.

Encoding Supercombinators. For a given rule $sc = (f_{src}, a_1, \ldots, a_c, a, f_{dest}) \in SC$, let $\mathit{Passive}(sc) = \{i \in \{1, \cdots, c\} \mid a_i = \epsilon$, i.e., P_i is not involved in $sc\}$. Let $u = (u_1, \ldots, u_c)$ be a vector of (supercombinator-independent) Boolean variables. We denote:

$$\mathit{lit}(u_i) = \begin{cases} u_i & \text{if } P_i \text{ is not involved} \\ \neg u_i & \text{if } P_i \text{ performs a visible event or a } \tau \end{cases}$$

Note that a process might be switched on in a format and still be passive in a certain supercombinator in this format. Hence, we cannot use the format to conclude which processes are passive in a supercombinator.

Let f and f' be two vectors of Boolean variables for encoding the source and destination format of a rule. Let $sc = (f_{src}, a_1, \ldots, a_c, a, f_{dest}) \in SC$. Then, $sc(y_1, \ldots, y_c, y, u, f, f') =$

$$\bigwedge_{i \notin \mathit{Passive}(sc)} (a_i(y^i) \ \wedge \ \neg u_i) \ \wedge \bigwedge_{i \in \mathit{Passive}(sc)} u_i \ \wedge \ a(y) \ \wedge \ f_{src}(f) \ \wedge \ f_{dest}(f').$$

Hence, in an encoding of a supercombinator, we indicate a passive process P_i just by affirming a single Boolean variable u_i. We call u_i a *trigger*. For non-passive processes, we also encode the event that the process performs. The encoding of all supercombinators in all formats now becomes the following:

$$SC(y^1, \ldots, y^c, y, u, f, f') = \bigvee_{sc \in SC} sc(y^1, \ldots, y^c, y, u, f, f').$$

Encoding a Transition of the Concurrent System. Let for $i \in \{1, \cdots, c\}$,

$$\psi_i(x^i, x^{i'}, y^i, u_i) := \text{if } u_i \text{ then } (x^i = x^{i'}) \text{ else } T^i(x^i, y^i, x^{i'}),$$

where $x^i = x^{i'}$ is the short for $\bigwedge_{j=1}^{n_i}(x^i_j \Leftrightarrow x^{i'}_j)$. The intuition behind a ψ_i is that, if P_i does not participate in a transition of the entire system, i.e., P_i is not involved in a supercombinator, P_i remains in the same state within its own labelled transition system OS^i. Otherwise, P_i progresses with respect to its transition relation T^i. Expressed as a Boolean formula,

$$\psi_i \equiv (u_i \ \wedge \ (x^i = x^{i'})) \ \vee \ (\neg u_i \wedge T^i(x^i, y^i, x^{i'})).$$

We define a predicate $T^{System(c)}$ which is true precisely for the transitions of the overall system:

$$T^{System(c)}(x^1, \cdots, x^c, x^{1'}, \cdots, x^{c'}, y^1, \cdots, y^c, y, u, f, f') =$$
$$= \bigwedge_{i=1}^{c} \psi_i(x^i, x^{i'}, y^i, u_i) \ \wedge \ SC(y^1, \cdots, y^c, y, u, f, f').$$

Encoding Fixed-Length Executions of the Concurrent System. Within the BMC framework, let k be the maximal bound for the length of the counterexamples we are looking for. Then:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{Paths}(\mathit{System}(c),k)(} \\
\text{// variables for } P_1 & x_0^1,\dots,x_k^1,y_1^1,\dots,y_k^1,u_1^1,\dots,u_k^1 \\
\text{// variables for } P_2 & x_0^2,\dots,x_k^2,y_1^2,\dots,y_k^2,u_1^2,\dots,u_k^2 \\
\dots & \dots \\
\text{// variables for } P_c & x_0^c,\dots,x_k^c,y_1^c,\dots,y_k^c,u_1^c,\dots,u_k^c \\
\text{// variables for the traces of the system} & y_1,\dots,y_k, \\
\text{// variables for the formats in the rules} & f_0,\dots,f_k)
\end{array}$$

$$\begin{array}{ll}
= & \text{// processes start from their initial states and the initial format is Format[0]} \\
& \bigwedge_{j=1}^{c} I^j(x_0^j) \wedge\ I^f(f_0) \wedge \\
& \text{// supercombinators as transitions at each of the } k \text{ steps} \\
& \bigwedge_{i=1}^{k} SC(y_i^1,\dots,y_i^c,y_i,u_i^1,\dots,u_i^c,f_{i-1},f_i) \wedge \\
& \text{// the idea of the } \psi \text{ formulas: either transitions or wait, depending on supercombinators} \\
& \bigwedge_{\substack{j=1,\dots,c \\ i=1,\dots,k}} ((u_i^j\ \wedge\ (x_{i-1}^j = x_i^j))\ \vee\ (\neg u_i^j\ \wedge\ T^j(x_{i-1}^j,y_i^j,x_i^j))) \\
= & \\
& I^{\mathit{System}(c)}(x_0^1,\dots,x_0^c,f_0) \wedge \\
& \bigwedge_{i=1}^{k} T^{\mathit{System}(c)}(x_{i-1}^1,\dots,x_{i-1}^c,x_i^1,\dots,x_i^c,y_i^1,\dots,y_i^c,y_i,u_i^1,\dots,u_i^c,f_{i-1},f_i)
\end{array}$$

3.5 Implementation Details

Our prototype tool SymFDR is written in C++ and uses FDR as a shared library for manipulating CSP processes. The current implementation of SymFDR supports refinement checking systems with a single format only. However, we do not anticipate any problems generalising the problem to a multi-format setting. Moreover, most practical cases are also single-format or can be easily rewritten in this form.

3.5.1 BMC

In our BMC framework, we support three modes of state space traversal—forward (starting from the initial state), backward (starting from an error state) and simultaneous forward/backward mode.

In the original version of BMC, the system is unwound step by step until the bound k is reached. Despite the recent advances in SAT solvers' learning capabilities and incremental SAT solving, we have observed that the bottleneck of the bounded refinement procedure is the SAT solver. Therefore, we allow unfolding a configurable number i of steps of the process *Refinement* before running the SAT solver, provided that the system never deadlocks on an erroneous trace, which we guarantee by the watchdog construction (see Section 3.3.2.2). The SAT solver is then used to check whether *Refinement* can reach *Watchdog*'s *sink* state in any of its last i unwindings. If so, we have found a counterexample, otherwise we continue iterating until reaching the configured bound k. We refer to the value of i as *SAT frequency*. We believe that this multi-step approach works well because the SAT solver typically finds it much easier to find a satisfying assignment, if there is any, than to prove unsatisfiability, given CNF formulas with comparable size and structure. Hence, we trade off the shortness of the reported counterexample for efficiency.

3.5.2 k-induction

We have implemented the "Zig-Zag" and "Dual" temporal k-induction algorithms [ES03b]. The difference is that the "Dual" algorithm makes use of separate SAT solvers for the base case and the induction step, aiming to optimise the incremental SAT interface. For k-induction, SymFDR supports both forward and backward traversal, yielding four algorithms in total: "Zig-Zag" forward, "Zig-Zag" backward, "Dual" forward and "Dual" backward. For all four algorithms we have observed that the induction step is checked much faster than the base one, opposite to what reported in [ES03b, SSS00]. In case of unsatisfiable instances, we have also observed that if the length of the longest possible counterexample is known in advance, often iterating the BMC algorithm up to this length produces better results than k-induction.

3.5.3 SAT

SymFDR supports both binary and one-hot encoding of state spaces, though we have observed that for our test cases binary encoding scales much better. One-hot encoding usually yields CNF instances with smaller number of clauses but substantially higher number of

Boolean variables which seems to burden the SAT solver. We construct the Boolean formulas directly in negation normal form and, consequently, transform them into equisatisfiable formulas in CNF using the optimised one-sided Tseitin encoding [BKWW08, BHvMW09], which we described in Section 2.1.4.

Currently, SymFDR supports MiniSAT 2.0 [EB05], PicoSAT 846 [Bie08] and ZChaff [MMZ$^+$01], all working in incremental mode. For our test cases, we have found MiniSAT to be the most efficient and all quoted results use MiniSAT. We exploit MiniSAT's incremental interface in a way similar to [ES03b]. For our larger test cases, we also observed that MiniSAT finds a counterexample faster if we configure it to keep a smaller number of learned clauses (learntsize_factor = 0.2, learntsize_inc = 1.02) and restart more frequently (restart_inc = 1.1). We also implemented adding unit learned clauses explicitly, as suggested in [ES03a], in conjunctions of multiple ones. Using positive polarity in decision heuristics also produced much better results, as well as freezing and then defreezing state and format variables at each step to avoid variable elimination.

SymFDR also supports strategies for restricting the decision variables to the input ones [Sht00], incorporating PicoSAT's restarting scheme and phase saving strategy [Bie08] in MiniSAT, etc.

3.6 Experimental Results

In this section, we analyse the performance of SymFDR on a small number of case studies. We compare it to the performance of FDR 2.83, FDR used in a non-standard way, PAT 3.2.2 [SLD08], and, in some cases, direct SAT encodings, NuSMV 2 [CCG$^+$02] and Alloy Analyzer 4.1.10 [Jac06]. All SAT-based experiments use MiniSAT although SymFDR and the direct SAT encoder build upon MiniSAT version 2.0, while Alloy and NuSMV exploit the earlier version 1.14. All tests were performed on a 2.6 GHz PC with 2 GB RAM running Linux, except the test marked with a *, which was performed on a 4-GB-RAM PC running Linux.

3.6.1 Comparison Tools

FDR-Div. The main search strategy for FDR is BFS [Ros94] because this has the combined advantages of always finding a shortest counterexample and of enabling implementations that work comparatively well on virtual memory. However, the strategy for discovering divergences is based on DFS. In test cases where it is likely that there are a good number of counterexamples, but that all of them occur comparatively deep in the BFS, there is good reason to use a bounded DFS (BDFS) algorithm to search for them, so that only error states reachable in less than some fixed number N of steps are reached. BDFS will quickly get to the depth where counterexamples are expected without needing to enumerate all of the levels where they are not. Provided that the counterexamples have something like a uniform distribution through the order in which the DFS discovers them, we can expect one to be found after searching through approximately $S/(C+1)$ states, where S is the total number of states and C is the number of counterexamples.

FDR does not implement such a strategy directly. It was, however, observed a number of years ago by Roscoe and James Heather that it is possible to use a trick that achieves the same effect using the present version of the tool. That is, arrange (perhaps using a watchdog) a system P' that performs only up to N events of the target implementation process P and then performs an infinite number of some indicator event when a trace specification is breached. Provided P is itself divergence-free, we then have that $P' \setminus \Sigma$ can diverge precisely when P violates the specification. FDR searches for this divergence by DFS.

This approach is particularly well suited to CSP codings of puzzles, since it is frequently known *ab initio* how long a counterexample will be, and the usual CSP coding uses the repeatable event *done* to indicate that the puzzle has been solved. The columns labelled FDR-Div in Tables 3.1 and 3.2 report on the result of using this technique. In several ways this method is more similar to approach of PAT and SymFDR than the usual FDR approach. As is apparent from the experiments, there seems to be a large element of luck in how fast this approach is, possibly based on how close the path followed by the DFS is to a counterexample.

PAT. PAT [SLD08] is a model checker of a version of CSP enhanced with shared variables. Despite the BMC attempt [SLDS08], PAT is at present a fully explicit checker. In addition to LTL model checking, PAT supports CSP refinement checking which it performs in a way similar to FDR although using DFS (instead of BFS), normalisation of the specification on-the-fly, partial-order reductions, counter abstraction, symmetry reduction, etc. In the test cases quoted here, the specification is given as a reachability property on the values of the shared variables, as modelled in the benchmarks available with the tool. The reachability algorithm is based on DFS and state hashing is applied for compact state-space representation.

NuSMV. NuSMV [CCG$^+$02] is a symbolic model checker verifying SMV against CTL properties using BDDs. The BMC framework of NuSMV, which we refer to as NuSMV-BMC, uses specifications written in LTL.

Alloy Analyzer. Alloy Analyzer [Jac06] is a fully-automatic tool for finding models of software systems designed in the lightweight Alloy modelling language. Alloy Analyzer could be considered a BMC checker due to its searching for a model only up to a certain scope and generating the model, if existing, using SAT-solving techniques.

Direct SAT Encodings. We believe that experimenting with direct SAT encodings of problems will offer guidance for optimising the translation of CSP to logic. For example, the chess knight test case suggests that a shorter chain of inference for high-level actions might be beneficial.

3.6.2 Test Cases

3.6.2.1 Instances with Counterexamples

In this section, we consider test cases with counterexamples and therefore exploit the BMC framework. The results are summarised in Tables 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6. The last column titled $\sharp$ lists the length of counterexamples.

First, we consider the peg solitaire puzzle [Ros98], performing experiments on a chain of soluble boards with increasing level of difficulty. In the initial configuration, the board has all slots but one occupied by pegs. The only allowed move in the game is a peg hopping over another peg and landing on an empty slot. The hopped-over peg is then removed from the board. The objective of the game is ending up with a board with a single peg positioned on the slot which had been initially empty. The length of any solution of the puzzle is equal exactly to the number N of pegs on the initial board—a hop event for $(N-1)$ pegs followed by an event *done* signifying a valid solution of the puzzle. The results are summarised in Table 3.1. The experiments indicate that, for $N \geq 26$, SymFDR clearly outperforms FDR. The performance of SymFDR and PAT seems to alternate. The explicit DFS-based tools FDR-Div and PAT seem to perform quite unevenly, signifying the importance of luck. We have also observed that in cases where a counterexample does not exist, FDR's BFS strategy outperforms the DFS-based tools PAT and SymFDR. To give an idea of the size of the SAT instances that SymFDR generates, for $N = 32$, the instance originally contains 314 567 clauses using 38 034 variables. MiniSAT learns extra 132 451 clauses and finds a satisfying assignment after 132 056 844 propagations. In comparison, FDR finds a solution after traversing 187 000 000 states.

Table 3.1: Performance comparison: peg solitaire with N pegs ($\sharp = N$)

N	FDR	Time (sec.)				SAT	$\sharp$
	$\sharp$ states checked	FDR	FDR -Div	PAT	SymFDR	freq.	
20	41 703	0	0	5.17	5.75	10	20
					18.14	20	
23	411 976	5	0	1.83	13.1	12	23
					7.69	23	
26	4 048 216	72	0	6.23	22.81	13	26
					21.47	26	
29	28 249 254	581	1	70.64	34.35	15	29
					17.35	29	
32	—	—	5	7.08	147.68	16	32
	187 000 000*	2 640*			66.3	32	
35	—	—	1 485	484.25	214.63	18	35
					90.97	35	
38	—	—	43	—	182.91	19	38
41	—	—	4	358.69	325.59	41	41

Our second test case is the chess knight tour. A knight is placed at position $(1,1)$ on an empty chess board of size $N \times N$. The objective is covering all squares of the board by visiting each square exactly once. Similarly to peg solitaire, a solution is generated as a counterexample to a specification asserting that the event *done* is never communicated. The length of a possible solution is $N^2 + 1$. The results are presented in Table 3.2. For

$N = 5$, FDR generates a counterexample faster, but, for $N = 6$, SymFDR found a solution in approximately 5 minutes, while FDR crashed after an hour and a half of state-space exploration. For this test case, we have observed that restricting the decision variables in MiniSAT to the input ones enhances the performance of SymFDR, especially for $N = 7$ where the reduction factor is over 20. Hence, for $N = 7$, the performance of the general tool SymFDR comes close to the performance of the problem-specific SAT encoder for the chess knight tour.

Table 3.2: Performance comparison: chess knight tour on a $N \times N$ board ($\sharp = N^2 + 1$)

N	FDR	Time (sec.)					SAT	$\sharp$
	$\sharp$ states checked	FDR	FDR -Div	PAT	Direct SAT	SymFDR	freq.	
5	508 451	3	0.147	0.53	8.5	8.15	13	26
6	$>$ 120 000 000	—	18	17.17	125.3	298.64	19	37
7	—	—	—	12.86	1 138.0	1 326.18	25	50

Table 3.3: Performance comparison: Hamiltonian cycle on a $N \times M$ grid ($\sharp = N \times M + 1$)

N	FDR	Time (sec.)			SAT	$\sharp$
	$\sharp$ states checked	FDR	FDR -Div	SymFDR	freq.	
4×4	2 518	0	0	0.83	9	17
4×5	14 368	0	0	1.98	11	21
5×6	1 219 416	21	33	32.9	16	31
				19.27	31	
6×6	18 115 326	243	630	43.54	19	37
6×7	$>$ 115 000 000	$>$ 3 600	$>$ 3 600	228.88	22	43

We have observed similar results with the test cases of finding a Hamiltonian path on an $N \times M$ grid (Table 3.3) and the lights-off puzzle (Table 3.4). The lights-off puzzle starts with an $N \times N$ board with all lights initially on. The aim is to reach a configuration with all lights switched off having in mind that upon triggering any light switch, the switch to the right, left, below and above is also triggered. This test case illustrates the difference that search mode can make in SymFDR. We remark that for every N, if using inductive normal compression, FDR can actually obtain a state space consisting of a single state which it can verify in 0 seconds.

Table 3.4: Performance comparison: lights off puzzle on a $N \times N$ board ($\sharp$ unpredictable)

N	FDR	Time (sec.)					SAT	$\sharp$
	$\sharp$ states checked	FDR	PAT	SymFDR fw	SymFDR bw	SymFDR fw/bw	freq.	
2	16	0		0.05	0.05	0.06	5	5
3	382	0		0.16	0.14	0.19	6	6
				0.23	0.24	0.32	5	
4	2 084	0		0.26	0.27	0.33	5	5
5	8 388 608	537	$> 7\ 600$	—	23.45	655.33	8	16
6	—	out of mem		—	—	—	15	28

The fifth test case—the classical puzzle of towers of Hanoi, aims primarily at comparing SymFDR with other SAT-based bounded checkers such as NuSMV and Alloy Analyzer. The results are summarised in Table 3.5. NuSMV-BMC and SymFDR seem to be competitive, both outperforming Alloy Analyzer. SymFDR working in simultaneous forward/backward mode outperforms NuSMV-BMC. However, all non-SAT tools—the explicit ones FDR and PAT and the BDD-based NuSMV—are clearly orders of magnitude more efficient than the SAT-based ones. We remark, though, that all solutions for the puzzle generated by PAT are longer than 1000 moves, even when $N = 5$, when the shortest solution is of length 32. When configuring PAT to report the shortest witness trace, we obtain the results quoted in the column labelled "PAT short". In this case, the performance of PAT worsens fast, falling behind SymFDR for $N = 7$ and running out of memory for $N = 8$.

Table 3.5: Performance comparison: Hanoi towers with N disks ($\sharp = 2^N$)

N	Time (sec.)								SAT	$\sharp$
	FDR	PAT	PAT short	Nu SMV	SymFDR fw	SymFDR fw/bw	Alloy	NuSMV -BMC	freq.	
5	0.19	0.21	0.98	0.43	4.9	3.6	11	2.2	16	32
6	0.20	1.18	10.15	0.66	27.3	21.6	327	34.9	32	64
7	0.16	2.18	202.4	0.17	182.7	173.3	21 537	1 865	64	128
8	0.18	15.20	—	0.29	3 114.1	2 035.1	—	2 218	128	256

Our final Table 3.6 summarises results obtained while running SymFDR on CSP scripts generated by *Casper* [Low98]—a well-known tool for analysing and verifying the correctness of security protocols, underlying the discovery of an attack on the Needham–Schroeder public key protocol in 1995 [Low95] and the verification of correctness of a fixed version of it in 1996 [Low96]. Casper takes a big advantage of the partial-order-reduction function *chase* offered by FDR [Ros98], as is apparent from the comparison of the performance of FDR with and without it. For the Needham–Schroeder public key protocol (NSPK3), SymFDR is better than FDR without *chase* but worse than FDR with *chase*. SymFDR finds those instances particularly hard because, on one hand, the state-space blow-up is enormous and, on the other hand, probably due to the great number of τ actions entangled into the state space, there are very few clauses learned and those clauses contain far too many literals (often over 1000). For these test cases, we have used MiniSAT configured with negative polarity and no decision on auxiliary variables.

Table 3.6: Performance comparison: (violated) security protocols

N	Time (sec.)					SAT	♯
	FDR *chase*	FDR no *chase*	SymFDR fw	SymFDR bw	SymFDR fw/bw	freq.	
NSPKP3	1	452	93.37	—	—	7	14
TMN1	0	0	34.82	36.89	30.73	7	7
Andrew	0	0	4.35	4.03	3.97	8	8

3.6.2.2 Instances without Counterexamples

In this section we focus on the performance of SymFDR using k-induction (see Table 3.7). For each of the four algorithms, "Zig-Zag" forward, "Zig-Zag" backward, "Dual" forward and "Dual" backward, we record the time in seconds and the step at which the algorithm terminates.

We consider the readers/writers test case, Milner's scheduler with and without compression, the bakery algorithm for mutual exclusion and the bully algorithm for leader election[1]. Besides manually generated scripts, we have experimented with scripts translated by Casper and SVA (Shared Variables Analyser) [Ros11b]—a front-end for FDR based on modelling

[1]Scripts and descriptions for all benchmarks are available from the website associated with [Ros11b].

concurrency using shared variables. We have also considered the effect of applying available FDR compression techniques to the CSP scripts and have analysed the impact of those techniques to the recurrence radii.

Table 3.7: Performance comparison: k-induction

Test	Zig-zag forward		Zig-zag backward		Dual forward		Dual backward	
	Time	k	Time	k	Time	k	Time	k
R/W 100	4.69	5	1.2	2	7.15	4	3.07	2
R/W 200	17.77	5	4.09	2	27.64	4	10.94	2
R/W 300	—	—	8.61	2	—	—	—	—
R/W 400	—	—	15.95	2	—	—	—	—
R/W 500	—	—	25.84	2	—	—	—	—
R/W 600	—	—	35.54	2	—	—	—	—
R/W 700	—	—	49.00	2	—	—	—	—
Milner 2	0.07	5	0.09	8	0.08	5	0.12	7
Milner 3	0.65	14	1.78	20	0.63	13	1.17	19
Milner 4	559.05	33	2181.07	50	376.03	32	57.86	49
Milner 2 leaf normal	0.03	3	0.04	4	0.02	3	0.05	4
Milner 3 leaf normal	0.17	9	0.08	6	0.21	8	0.13	6
Milner 4 leaf normal	0.67	13	0.19	8	0.64	12	0.27	8
Milner 5 leaf normal	6.17	20	0.38	10	6.55	19	0.48	10
Milner 6 leaf normal	306.84	29	0.75	12	225.15	28	0.88	10
Milner 10 leaf normal	—	—	4.84	20	—	—	7.13	20
Milner 15 leaf normal	—	—	27.58	30	—	—	21.64	30
Milner 20 leaf normal	—	—	155.1	40	—	—	103.53	40
Bakery 2 (4), SVA hc	0.14	2	0.26	9	0.15	2	—	—
Bakery 2 (8), SVA hc	0.29	2	1.86	21	0.29	2	—	—
Bakery 2 (16), SVA hc	0.89	2	17.4	45	0.83	2	—	—
Bakery 2 (32), SVA hc	2.92	2	—	—	2.98	2	—	—
Bakery 2 (64), SVA hc	12.9	2	—	—	13.53	2	—	—
Bakery 3 (4), SVA hc	—	> 33	—	—	—	—	—	—
Bully 3 (1,2,3)	—	—	152.95	40	—	—	—	—
Bully 3 (1,3,7)	—	—	151.97	40	—	—	—	—

In our experience, the backward algorithm, aiming to reach the forward recurrence radius, often scales better than the forward one. We note that, due to concurrency, the completeness threshold blows up in all cases. Hence, successful performance mainly depends on whether the property is k-inductive or not, for some small value of k. For the bakery algorithm and Milner's scheduler, applying, respectively, hierarchical and leaf compression (see

Section 2.3.3.2), has proven to be beneficial for reducing the recurrence diameter, thereby significantly decreasing the termination step and, hence, improving the performance. For all four algorithms we have observed that the induction step is checked much faster than the base one, opposite to what reported in [ES03b, SSS00]. Hence, we also implemented versions of k-induction starting the iteration process from a step greater that zero, as suggested in [SSS00]. Opposite to what we expected, though, for our test cases this approach scales worse than the standard one. In case of unsatisfiable instances, we have also observed that if the length of the longest possible counterexample is known in advance, often iterating the BMC algorithm up to this length produces better results than k-induction due to tuning the SAT frequency and jumping multiple time steps at once.

3.6.2.3 Discussion

We can conclude that SymFDR is likely to outperform FDR in large tightly-coupled combinatorial problems for which a solution exists, the length of the longest solution is relatively short (growing at most polynomially) and is predictable in advance. In those cases, we can fix the SAT frequency close to a sizeable divisor of this length and thus spare large SAT overhead. The search space of those problems can be characterised as very wide (with respect to BFS), but relatively shallow—with counterexamples with length up to approximately 50–60. We suspect that problems with multiple solutions also induce good SAT performance. The experiments with the towers of Hanoi suggest that SAT-solving techniques offer advantages up to a certain threshold and weaken afterwards.

SymFDR in k-induction mode works reasonably well for small test cases, especially if the property is k-inductive for some small value of k. However, for larger test cases, SymFDR does not scale very well as the completeness threshold becomes too large due to concurrency. In all cases considered in Table 3.7, FDR is considerably faster.

3.7 Conclusion and Future Work

In this chapter we have demonstrated the feasibility of integrating SAT-based BMC and k-induction in FDR, and more specifically, exchanging the expensive explicit state-space traversal phase in FDR by a SAT check in SymFDR. On some test cases, such as complex combinatorial problems, SymFDR's performance is very encouraging, coping with problems that are beyond FDR's capabilities. In general, though, FDR usually outperforms SymFDR, particularly when a counterexample does not exist. We plan to further investigate and try to gain insight about the classes of problems that are tackled more successfully within the BMC framework.

We envision several directions for future work.

We intend to extend the BMC framework in SymFDR to make it applicable to the stable failures and failures-divergences models as well. This will involve extending the encoding of CSP processes with information about minimal acceptances and divergences. Alternatively, we could try and tailor the translation from liveness to safety properties detailed in [BAS02].

Our future plans mostly revolve around investigating alternative methods for obtaining complete SAT-based refinement checking, two approaches for which we describe below.

3.7.1 Interpolation Techniques

We plan to implement and experimentally evaluate McMillan's algorithm which combines SAT and Craig interpolation techniques [Cra57] to yield complete unbounded refinement checking [McM03, McM06]. This approach relies on a SAT solver being capable of generating a resolution proof of unsatisfiability for an unsatisfiable SAT instance. A *Craig interpolant* [Cra57] of two sets of clauses A and B with $A \cup B$ being unsatisfiable is a Boolean formula P such that A implies P, $P \wedge B$ is unsatisfiable, and P only refers to the common variables of A and B [Cra57, McM03].

In the context of BMC, a proof of unsatisfiability is a proof that there are no counterexamples of length less than or equal to k. An unsatisfiable BMC SAT instance for a given bound k and Craig interpolation can be combined to calculate an overapproximation of the forward image operator. A is taken to be the set of clauses corresponding to the initial condition I and the unfolding of the first transition. B represents the unfolding of the rest $(k-1)$ transitions and the error condition F.

$$\underbrace{I(s_0) \ \wedge \ T(s_0, s_1)}_{A} \ \wedge \ \underbrace{T(s_1, s_2) \ \wedge \ \dots T(s_{k-1}, s_k) \ \wedge \ \bigvee_{i=1}^{k} F(s_i)}_{B}$$

Then, since A implies P, the interpolant P is an overapproximation of the set of states reachable in one step from the initial states. Furthermore, since $P \wedge B$ is unsatisfiable, no state satisfying P can reach states satisfying F in $(k-1)$ steps. Therefore, P is an underapproximation of the set of states unreachable backwards from F in $(k-1)$ steps. In addition, P refers only to the variables used for encoding s_1.

Interpolation methods have been successfully applied for efficiently deriving inductive program invariants [McM06, McM03]. Those invariants can be used for establishing safety properties (or liveness properties if translated into safety first [BAS02]). Craig interpolation exploits the ability of SAT solvers to narrow proofs of unsatisfiability to only relevant variables to obtain an abstract image operator out of an unsatisfiable BMC instance for a given value k, as described above. Incorporating this operator into a fixed-point computation results into a decision procedure that only explores those parts the state space that are relevant to the specification property under consideration. If the fixed-point computation for the particular k fails to establish the correctness of the property and detects a counterexample that relies on the overapproximation, the process is restarted with BMC instances for values greater than k, which generate more precise overapproximations of the image operator by increasing the minimal distance to error states.

The combination of SAT and interpolation has proven to be more efficient for positive hardware BMC instances (instances with no counterexamples) than other SAT approaches [McM03, ADK$^+$05]. The completeness threshold in this case is the backward radius of the state space which is smaller than its backward recurrence radius, as is the case with temporal k-induction. Moreover, experimental results have shown that, in practice, the algorithm often converges substantially faster, for bounds considerably smaller than the backward radius. In addition, the interpolation algorithm allows jumping multiple time frames at once and hence permits tuning the SAT frequency. The BMC framework presented in this chapter is the foundation we will build upon.

We also plan to experiment with applying interpolants of different quality and strength as described in [DKPW10]. For both temporal k-induction and interpolation-based refinement checking, we intend to investigate techniques for invariant strengthening, which would be helpful for decreasing the corresponding completeness thresholds [BHvMW09]. For tackling the challenges arising from concurrency, we speculate that exploiting the combination of interpolation methods and partial-order reductions [CGP99, Pel98] would prove beneficial.

3.7.2 IC3 and PDR Frameworks

A new approach for generating invariants that strengthen a given safety property under scrutiny was relatively recently proposed [BM08, EMB11, Bra11, Bra12], alternative to Ken McMillan's interpolation algorithm. In literature, it is often referred to as IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) [Bra11, Bra11] or PDR (property-driven reachability analysis) [EMB11]. As opposed to the temporal k-induction and interpolation algorithms, which are monolithic in their aim to produce a single general inductive invariant strengthening the safety property, IC3 and PDR are incremental in their approach—they incrementally generate a series of simpler auxiliary invariants each of which is inductive relative to the previous ones.

Similarly to the interpolation-based method, IC3 operates by incrementally and symbolically generating a sequence of sets of states $\langle F_0, \ldots, F_k \rangle$ for increasingly larger $k \in \mathbb{N}$, where for every $0 \leq i \leq k$, F_i is an overapproximation of the sets of states reachable in i or less steps from the initial state and F_i only contains states that are at least $(k - i + 1)$ steps away from error states. Each F_i is maintained as a set of clauses and for all $0 \leq i < k$, $\mathsf{clauses}(F_{i+1}) \subseteq \mathsf{clauses}(F_i)$ always holds. The sequence $\langle F_0, \ldots, F_k \rangle$ is constructed to always satisfy the following properties (given a model $\langle I, T \rangle$ and a safety property P):

- $I \Rightarrow F_0$
- For all $0 \leq i < k$, $F_i \Rightarrow F_{i+1}$ (which is implied by $\mathsf{clauses}(F_{i+1}) \subseteq \mathsf{clauses}(F_i)$)
- For all $0 \leq i \leq k$, $F_i \Rightarrow P$
- For all $0 \leq i < k$, $F_i \wedge T \Rightarrow F'_{i+1}$, which implies that F_{i+1} is inductive relative to F_i.

In contrast to the interpolation algorithm, which restarts the fixed-point computation with a more precise image operator every time when it detects a counterexample that can possibly be spurious, IC3 employs an algorithm that moves both back and forth along the sequence of approximations. Major iterations of the algorithm increase the frontier step k, while minor iterations refine i-step approximations, thus reusing and strengthening lemmas gathered so far. The refinement is performed on the syntactic level by adding a lemma (clause) c to all $F_0, \ldots, F_j$ for some $j \leq k$.

Another crucial advantage of IC3 is that it requires just a single unfolding of the transition relation, as opposed to the k-induction and interpolation algorithms that build upon

BMC and need to symbolically unroll the transition relation multiple times. Effectively, IC3 operates by repeatedly defining and fulfilling proof obligations that are generated on demand out of counterexamples of inductiveness [BM08]. If the induction consecution step $F_k \wedge T \Rightarrow P'$ for the frontier F_k fails to hold, a state $s \in F_k$ is identified that is just a single step away from an error state [Bra11]. A proof obligation is then adjoined at frame k, aiming to prove that s is never reachable from the initial state in k or less steps (or even better, at all). For proving the new obligation, the algorithm may need to inductively refine a series of approximations $F_j, \ldots, F_k$ for some $j \leq k$ by adding/proving/refuting more obligations at backward steps. If the algorithm fails to fulfill an obligation for the initial time frame, the property is violated, in which case a counterexample can be reconstructed by suitable bookkeeping. If at some point $\neg s$ is proven to be inductive relative to F_k, the inductiveness of P relative to F_k is tested again. If in subsequent tests the consecution step $F_k \wedge T \Rightarrow P'$ succeeds, a new frontier frame $F_{k+1} = P$ is appended. The algorithm converges whenever for some $k \in \mathbb{N}$, F_k is itself inductive, i.e., whenever $\mathsf{clauses}(F_{i+1}) \equiv \mathsf{clauses}(F_i)$.

The completeness threshold of the IC3 algorithm is the number of states in the system, which is greater than the completeness thresholds of both the k-induction and interpolation algorithms. However, empirically, IC3 has been found to work extremely well on industrial benchmarks, outperforming interpolation-based implementations [BM08] and finding counterexamples buried deep in the system. On one hand, this is due to the mitigation of the state-space explosion problem due to a single unrolling of the transition relation. On the other hand, IC3 employs various syntactic and semantic optimisations, e.g., inductive generalisation of the counterexample of inductiveness, propagating clauses forward by stepwise approximations, etc. The algorithm also benefits from low memory requirements and from the simplicity of its SAT queries. In addition, IC3 is readily amenable to parallelisation [BM08] by employing multiple SAT solvers working in parallel that exchange information about proven lemmas.

The novel IC3 framework is still in its early stage of development but has already demonstrated a lot of promise and is currently an extremely active area of research. To the best of our knowledge, it has not be investigated in the context of concurrent software systems yet, something that we plan to do in the context of CSP.

Chapter 4

Static Analysis for Livelock Detection

4.1 Introduction

It is standard in process algebra to distinguish between the visible and invisible (or silent) actions of a process. The latter correspond to state changes arising from internal computations such as resolving of nondeterminism, unfolding of a recursion, abstraction of details. Their occurrence is silent and is not detectable or controllable by the environment. A process is said to *diverge* or *livelock* if it reaches a state from which it may forever compute internally through an infinite sequence of invisible actions, thereby becoming unresponsive or 'hanging'.

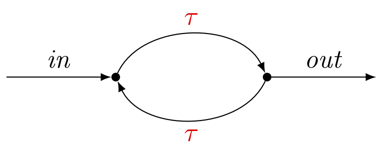


Figure 4.1: Livelock

Livelock is usually a highly undesirable 'feature' of a process, described in the literature as "even worse than deadlock, in that like an endless loop it may consume unbounded computing resources without achieving anything" [Hoa85]. Livelock invalidates certain analysis methodologies, e.g., it signifies lack of progress, and is often symptomatic of a bug in the modelling.

4.1.1 Sources of Livelock

The possibility of writing down divergent processes arises from the presence of two crucial constructs—recursion and hiding. Hiding is a key device for abstraction—it conceals 'internal chatter' and other irrelevant computations by converting visible actions into invisible ones. Recursion is what gives expressive power to a process algebra as it contributes Turing completeness.

We distinguish two nontrivial patterns through which a process may livelock. In the first place, livelock may be introduced if hiding an event or a set of events that can occur uninterruptedly and infinitely often to the exclusion of all others. Alternatively, a process may livelock owing to the presence of an unguarded recursion. Roughly speaking, the latter means that the process may recurse without communicating a visible event first.

4.1.2 Complexity of Establishing Livelock Freedom

In the context of the process algebra CSP, the problem of determining whether a process may livelock is in general undecidable[1]. Furthermore, even for finite-state processes, if adopting exhaustive state-space exploration techniques, establishing livelock freedom can be expensive due to concurrency and large data domains.

One way to check a process for divergence is to search for reachable cycles of silent actions in its state space, which is a labelled transition system built from the operational semantics. Assuming the underlying graph is finite, this can be achieved by calculating its strongly connected components, using, e.g., Tarjan's algorithm [CLRS01]. The latter can be carried out in time linear in the size of the graph, which may, however, be exponential (or worse) in the syntactic size of the term describing the process.

In the current implementation of FDR, the check for divergence is performed during the refinement checking phase by triggering a local DFS search for τ actions each time when encountering an unstable state. Hence, as pointed out in Section 2.3.2, refinement checking in the failures-divergences model is much more expensive than refinement checking in the stable-failures model, despite the fact that for livelock-free implementations the refinement checks in the two models yield the same results and counterexamples, if any.

[1]For example, CSP can encode counters, and is therefore Turing-powerful.

4.1.3 Our Proposal: Static Analysis

We propose a static livelock analysis framework, which conservatively flags potentially divergent CSP processes [OPRW11, OPRW13].

By static we mean that the algorithm works on the syntax of a process and never builds or explores the whole of its state space, as opposed to methods underlying model checking or refinement checking. In this way we circumvent the state-space explosion problem.

However, the algorithm is conservative, which means that it is sound but not complete. If it reports that a process is livelock free, then this is guaranteed to be the case. The alternative is ‘potential livelock’, which indicates an inconclusive result and might be a false positive. In this way we basically trade accuracy for speed.

General Framework. Our general framework can handle the widest variety of CSP processes, including infinite-state ones. The analysis is based on reasoning about fixed points in terms of metric spaces, as well as on keeping track and overapproximating the fair sets of events of a process. We introduce a new family of metrics parametrised by sets of visible events. Our framework automatically generates a sound but incomplete set of metrics that guarantee that recursive processes have unique fixed points, and furthermore, that those unique fixed points are livelock-free. The algorithm employs a collection of rules, based on the inductive structure of terms. The rules principally investigate the interaction of hiding, renaming and recursion and the way they may conspire to introduce livelock. The analysis (see Section 4.6) naturally divides into two parts according to the two nontrivial sources of livelock outlined above.

Framework for Structurally Finite-State Processes. In practice, the majority of processes are finite state and for those we offer a simpler, more precise and more efficient framework. Using compositional rules, we generate a pair consisting of a livelock flag and a collection of fair/co-fair pairs of sets of events, the combination of which can be viewed as an abstraction of the system preserving livelock freedom. The algorithm also benefits from being able to identify the minimal closed sequential components and examine their transition systems in isolation. For those it computes exact abstractions and starts becoming conservative only in the compositional rules for handling compound CSP processes, thereby allowing more elaborate and finer data to be computed efficiently.

Our Tool SLAP. We have implemented both frameworks in a tool called slap, which is an acronym for Static Livelock Analyser of Processes. Computationally, the crux of our algorithms revolves around the generation and manipulation of sets. The algorithms fit very naturally into a symbolic paradigm; hence slap is fully symbolic. The choice of an underlying symbolic engine is configurable, with support for using a SAT engine (based on MiniSAT 2.0), a BDD engine (based on CUDD 2.4.2), or running a SAT and a BDD analyser in parallel and reporting the results of the first one to finish.

The symbolic implementation is particularly empowering in the framework for handling structurally finite-state processes and, more specifically, in its base case when dealing with minimal sequential processes. For those, we encode the exact computation of the collection of fair/co-fair pairs into a compact circuit of size polynomial in the syntactic description of the process in question. This circuit we then represent symbolically by either burning it into a BDD or turning it into an equisatisfiable Boolean formula intended as an input to a SAT solver.

By circumventing the state-space exploration, we obtain a static analysis algorithm which in practice tends to substantially outperform state-of-the-art model-checking tools such as FDR—see Section 4.9 for experimental comparisons.

Naturally, there is a trade-off between the speed and accuracy of livelock checking. It is not hard to write down processes which are livelock-free but which our analysis indicates as potentially divergent. However, when modelling systems in practice, it makes sense to try to check for livelock freedom using a simple and highly economical static analysis before invoking computationally expensive state-space exploration algorithms. Indeed, as Roscoe [Ros98, page 208] points out, the calculations required to determine if a process diverges are significantly more costly than those for deciding other aspects of refinement, and it is advantageous to avoid these calculations if at all possible. Hence, we have already integrated the framework for analysing structurally finite-state processes in FDR [AGL$^+$12], where it now constitutes an alternative back-end for establishing livelock freedom.

We note that our static livelock analysis is tailored to the setting of CSP, but the principles upon which our analysis is based should be transferable to other process algebras and other concurrent settings and formalisms, in general.

4.1.4 Related Work

The basic intuitions underlying our approach are fairly straightforward. In part they mirror the guardedness requirements which ensure that well-behaved CSP process equations have unique, livelock-free fixed points [Ros98, Chap. 8]. However, we extend the treatment of [Ros98] by allowing guarded recursions to include instances of the hiding operator. Incidentally, Milner's notion of guarded recursions in CCS is similarly restricted by the requirement that variables not occur inside parallel compositions [Mil89]. Complications arise mainly because we want to be able to fully incorporate hiding and renaming in our treatment, both of which can have subtle indirect effects on guardedness.

We note that the idea of guarded recursions is standard in process algebra. For instance, in Milner's framework, a variable is 'strongly guarded' in a given term if every free occurrence of the variable in the term occurs within the scope of a prefixing operator [Mil89]. This notion is introduced in order to justify certain proof principles, such as that guaranteeing the uniqueness of fixed points up to bisimilarity. Strong guardedness has also been extended to a calculus with hiding and action refinement [BG02]. A key difference between our approach and these notions is that we seek to guarantee livelock freedom, rather than merely the existence of unique fixed points.

In fact, there are few papers which deal with the problem of guaranteeing livelock freedom in the setting of concurrent process calculi.[2] The existing work on livelock freedom has mostly been carried out in the context of mobile calculi. [San02] presents an approach for guaranteeing livelock freedom for a certain fragment of the π-calculus. Unlike the combinatorial treatment presented here, this approach makes use of the rich theory of types of the π-calculus, and in particular the technique of logical relations. Another study of divergence freedom in the π-calculus appears in [YBH01], and uses the notions of graph types.

[2] In contrast, there are numerous works treating termination for the λ-calculus or combinatory logic [GLT88, Mit96, Gan80].

Note that CSP is predicated upon *synchronous* (i.e., handshake) communication. In terms of livelock analysis, different issues (and additional difficulties) arise in an asynchronous context (assuming unbounded communication buffers); see, e.g., [LcW06, LcW08].

Recent works in which CSP livelock freedom plays a key role include [Dim10] as well as [STW11, STW10]; see also references within.

4.2 Case Study: the Alternating Bit Protocol

In this section we briefly describe a network communication protocol called the alternating bit protocol (ABP), which we use as a test case. We also provide an abstracted version of the protocol that we use to illustrate our concepts.

4.2.1 Description and Implementation

Suppose we want to implement a reliable order-preserving message transfer from a point *Sender* to a remote point *Receiver* under the assumption that the communication media between those two remote points are error-prone. In particular, we will assume that:

1. The media are lossy, i.e., they can lose messages arbitrarily, provided that they do not lose an infinite number of messages in a row;
2. The media, however, cannot corrupt the contents of the messages, neither can they shuffle the order of messages in the flow.

There are a number of protocols that run correctly under those assumptions, one of which is the alternating bit protocol (ABP). It relies on the receiver end sending positive acknowledgments back to the sender upon successful reception of a message, as well as on both the receiver and the sender appending an extra control bit to every message that is sent across the lossy media. The bit alternates between 0 and 1 and, having that the order of messages is always preserved, proves sufficient to set a clear boundary between the transmission of subsequent messages.

The architecture of the system is depicted in 4.2, where E and F are the unreliable media used for sending messages and acknowledgments, respectively. Both the sender and the receiver are parametric in a control bit and, furthermore, the sender also holds the message it currently transmits to the receiver. Initially, we assume that *Sender* starts with

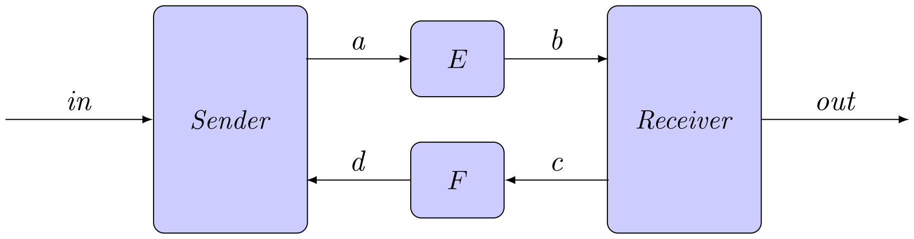


Figure 4.2: The alternating bit protocol

no messages pending (we use NULL) and a control bit valued 1. The process *Receiver* is initialised with a control bit valued 0. The protocol works as follows.

The sender (see Figure 4.3) intercepts a new message on its *in* channel, flips its control bit (to mark the message as new) and repeatedly sends the message tagged with the control bit to the receiver, until it obtains an acknowledgment marked with the same control bit. At this point, the sender restarts the protocol waiting for the next message to arrive. Acknowledgements that are received but not expected, i.e., those that are decorated with an inverted control bit, are simply ignored.

```
$Send(msg, bit)$ = (if ($msg$ == NULL)
                        then $in?x \longrightarrow Sender(msg, 1 - bit)$
                        else $a!bit!msg \longrightarrow Send(msg, bit)$)
                   $\Box$
                   ($d?ack \longrightarrow$ ( if ($ack == bit$)
                                  then $Send$(NULL, $bit$)
                                  else $Send(msg,\ bit)$))

$Sender = Send(\text{NULL}, 1)$
```

Figure 4.3: ABP: the sender process

When the receiver (see Figure 4.4) obtains a message marked with a control bit equal to its own, it flushes the message out to the environment via its *out* channel, flips its control bit and starts repeatedly sending acknowledgments back to the sender attaching the bit the message came with. The process is repeated until the receiver receives a new message, at which point the protocol is restarted. Subsequent messages with the same control bit are only acknowledged but not output to the environment.

$$\boxed{\begin{array}{l} \mathit{Receive}(\mathit{bit}) = b?\mathit{tag}?\mathit{msg} \longrightarrow (\text{if } (\mathit{tag} == \mathit{bit}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{then } \mathit{out}!\mathit{msg} \longrightarrow \mathit{Receive}(1 - \mathit{bit}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \mathit{Receive}(\mathit{bit})) \\ \qquad\qquad\qquad\quad \Box \\ \qquad\qquad\qquad\quad c!(1 - \mathit{bit}) \longrightarrow \mathit{Receive}(\mathit{bit}) \\ \\ \mathit{Receiver} = \mathit{Receive}(0) \end{array}}$$

Figure 4.4: ABP: the receiver process

Let us suppose that the lossy media E and F can nondeterministically lose up to N messages in a row. We can implement them as illustrated in Figure 4.5.

$$\boxed{\begin{array}{l} E(n) = a?\mathit{bit}?\mathit{msg} \longrightarrow (\text{if } (n == 0) \\ \qquad\qquad\qquad\qquad\qquad \text{then } b!\mathit{bit}!\mathit{msg} \longrightarrow E(N-1) \\ \qquad\qquad\qquad\qquad\qquad \text{else } (b!\mathit{bit}!\mathit{msg} \longrightarrow E(N-1) \sqcap E(n-1))) \\ \\ F(n) = c?\mathit{bit} \longrightarrow (\text{if } (n == 0) \\ \qquad\qquad\qquad\qquad \text{then } d!\mathit{bit} \longrightarrow F(N-1) \\ \qquad\qquad\qquad\qquad \text{else } (d!\mathit{bit} \longrightarrow F(N-1) \sqcap F(n-1))) \end{array}}$$

Figure 4.5: ABP: the lossy media

Then we can compose the whole system together as follows:

$$\mathit{System} = (\mathit{Sender} \underset{\{\mid a,d \mid\}}{\parallel} (E(N-1) \mid\mid\mid F(N-1)) \underset{\{\mid b,c \mid\}}{\parallel} \mathit{Receiver}) \setminus \{\mid a, b, c, d \mid\}$$

Notice that we hide all the communications over the lossy media at the top level. Indeed, since we are only interested in proving that messages from *Sender* eventually arrive at *Receiver* and do so in the same order they have been sent, the way that messages are propagated across the network is irrelevant. Hence, the only events we leave observable from outside are the messages transmitted through the channels *in* and *out*. The danger here is that the system might reach a configuration from which it spends the whole time in internal communications across the lossy media and never offers to input a message from *in* or output one from *out*. This scenario corresponds to a livelock, and in the way the system is designed currently, it is indeed possible, however undesirable it might be. For instance, *Receiver* can keep sending acknowledgments forever without *Sender* ever intercepting and sending a new message. Alternatively, a message can be transmitted and received forever without being acknowledged. Hence, even though the media E and F can lose only a finite number of messages in a row, it turns out that *System* still does not guarantee to deliver messages.

There are different strategies for enforcing conditions that eliminate the livelock and turn ABP into a reliable communication protocol. Some of those are fairness conditions that aim to eliminate the possibility that an endless message flow gets restricted only to the channels a and b or, alternatively, to c and d. For example we can impose the constraint that messages transmitted through the channels b and c strictly alternate:

$$\mathit{Alt} = b?\mathit{bit}?\mathit{msg} \longrightarrow c?\mathit{bit} \longrightarrow \mathit{Alt}$$

Then the process $\mathit{ASystem} = \mathit{System} \underset{\{\mid b,c \mid\}}{\parallel} \mathit{Alt}$ will satisfy our correctness criteria. Alternative strategies may add the constraints that the number of messages transmitted through the channel b and the number of messages transmitted through the channel c never differ by more than a constant, or that the number of messages transmitted through one of them in a row without using the other one is never greater than a constant.

4.2.2 An Abstracted Version of the ABP

The process *Send* (see Figure 4.6) attempts to send messages to itself infinitely often. Those messages, however, have to go through an unreliable network *Medium*, which may do an arbitrary (possibly infinite) number of *error* events before delivering the message back to *Send* in the form of an *out* event. We impose a fairness constraint *Fair* on *Medium*, forcing it to do at most a single error before delivering the message correctly, i.e., we require that every *error* event be immediately followed by an *out* event. We construct the system by putting the mutually-recursive processes *Send* and *Medium* in parallel with the process *Fair*, synchronising on the set of their shared events $\{\mathit{error}, \mathit{out}\}$ and hiding the *error* event at the top. The resulting process *System* is livelock-free and is, in fact, equivalent to the process $B_1 = \mathit{in} \longrightarrow \mathit{out} \longrightarrow B_1$, which implements a single-slot buffer. It is perhaps interesting to note that if we hide on top level the set of events $\{\mathit{error}, \mathit{out}\}$, instead of just $\{\mathit{error}\}$, the resulting system would still be livelock-free: every infinite execution of it would contain infinitely many occurrences of the event *in*.

$$\boxed{\begin{array}{l}
\mathit{Send} = \mathit{in} \longrightarrow \mathit{Medium} \\
\mathit{Medium} = \mathit{out} \longrightarrow \mathit{Send} \;\Box\; \mathit{error} \longrightarrow \mathit{Medium} \\
\\
\mathit{Fair} = \mathit{out} \longrightarrow \mathit{Fair} \;\Box\; \mathit{error} \longrightarrow \mathit{out} \longrightarrow \mathit{Fair} \\
\\
\mathit{System} = (\mathit{Send} \underset{\{\mathit{error},\mathit{out}\}}{\parallel} \mathit{Fair}) \setminus \{\mathit{error}\}
\end{array}}$$

Figure 4.6: ABP: an abstracted version

4.3 Syntax and Conventions

As we have seen so far, CSP processes are mostly defined via vectors of mutually-recursive equations. In this section we take a different view on recursive processes and represent them in functional form.

Similarly to Section 2.2.1, let Σ be a finite set of events, with $\checkmark \notin \Sigma$. We write $\Sigma^{\checkmark}$ to denote $\Sigma \cup \{\checkmark\}$ and $\Sigma^{*\checkmark}$ to denote the set of finite sequences of elements from Σ which may end with $\checkmark$. In the notation below, we have $a \in \Sigma$ and $A \subseteq \Sigma$. R denotes a binary renaming relation on Σ. The variable X is drawn from a fixed infinite set of process variables.

Definition 4.3.1. *CSP terms are constructed according to the following grammar:*

$$P ::= \mathit{STOP} \mid a \longrightarrow P \mid \mathit{SKIP} \mid P_1 \sqcap P_2 \mid P_1 \Box P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid$$
$$P_1 \,;\, P_2 \mid P \setminus A \mid P[\![R]\!] \mid X \mid \mu X \centerdot P(X) \mid \mathit{DIV}\ .$$

The informal description of all core CSP processes and operators is presented in Section 2.2.1. $\mu X \centerdot P(X)$ denotes a recursive process—to understand its meaning, consider the equation $X = P$ in terms of the unknown X. While this equation may have several solutions, it always has a unique least[3] such, written $\mu X \centerdot P(X)$.

A CSP term is *closed* if every occurrence of a variable X in it occurs within the scope of a μX operator; we refer to such terms as *processes.*

Definition 4.3.2. *A CSP process is a closed CSP term.*

We will denote by $\overline{\mathsf{CSP}}$ the set of all CSP processes and by CSP the set of all CSP terms, both open and closed.

[3] In a relevant partial order

Let us state a few conventions. When hiding a single event a, we write $P \setminus a$ rather than $P \setminus \{a\}$. The binding scope of the μX operator extends as far to the right as possible. We also often express recursions by means of the equational notation $X = P$, rather than the functional $\mu X \bullet P$. Let us remark that vectors of mutually recursive equations can always be converted to our present syntax that uses the fixed-point operator μ, thanks to Bekič's theorem [Win93, Chap. 10]. Accordingly, we shall freely make use of the vectorised notation as well.

In a nutshell [OPRW13], Bekič's theorem expresses fixed points of self-maps on the product space $X \times Y$ in terms of fixed points of self-maps on the respective components X and Y. For example, let us consider a mutually recursive process definition of the form $P = f(P, Q)$, $Q = g(P, Q)$. The idea is first to define a parametrised fixed point of g via the expression $\mu Y.g(X, Y)$, and then substitute in the expression for P, yielding $P = \mu X.f(X, \mu Y.g(X, Y))$. This process can be generalised to transform mutually recursive definitions of arbitrary dimension into expressions using only the single-variable fixed-point operator μ.

As an example, let us consider the mutually-recursive processes *Send* and *Medium* from Figure 4.6. The functional representation of the process *Send* will be the following:

$$\mu\, \mathit{Send} \bullet \mathit{in} \longrightarrow (\mu\, \mathit{Medium} \bullet \mathit{out} \longrightarrow \mathit{Send} \,\Box\, \mathit{error} \longrightarrow \mathit{Medium}).$$

4.4 Operational and Denotational Semantics

4.4.1 Operational Semantics

The rules presented in Section 2.2.4 allow us to associate to any CSP process a labelled transition system representing its possible executions. We say that a process *diverges* if it has an infinite path whose actions are exclusively τ's. A process is *livelock-free* if it never reaches a point from which it diverges.

4.4.2 Denotational Semantics

4.4.2.1 The Model $\mathcal{T}^{\Downarrow}$

In this section, we introduce the compositional model $\mathcal{T}^{\Downarrow}$ [Ros11b] in which a process P is identified by the pair $(\mathsf{traces}_{\perp}(P), \mathsf{divergences}(P))$. In this model, $\mathsf{traces}_{\perp}(P) = \mathsf{traces}(P) \cup$

$\mathsf{divergences}(P) \subseteq \Sigma^{*\checkmark}$ is the set of all finite visible-event traces that P may perform, and $\mathsf{divergences}(P) \subseteq \mathsf{traces}_\perp(P)$ is the set of traces after which it may diverge. Standard models of CSP also take account of the liveness properties of a process by modeling its *refusals*, i.e., the sets of events it cannot perform after a given trace. In our handling, we do not record refusals as they are orthogonal to the divergences of a process—see [Ros98, Sect. 8.4].

Following [Ros11b], we write $\mathcal{T}^\Downarrow$ to denote the set of pairs $(T, D) \in \mathcal{P}(\Sigma^{*\checkmark}) \times \mathcal{P}(\Sigma^{*\checkmark})$ satisfying the following axioms (where $\frown$ denotes trace concatenation):

1. $D \subseteq T$.
2. $s^\frown\langle\checkmark\rangle \in D$ implies $s \in D$.
3. $T \subseteq \Sigma^{*\checkmark}$ is non-empty and prefix-closed.
4. $s \in D \cap \Sigma^*$ and $t \in \Sigma^{*\checkmark}$ implies $s^\frown t \in D$.

Axiom 4 says that the set of divergences is postfix-closed. Indeed, since we are only interested in *detecting* divergence, we treat it as catastrophic and do not attempt to record any meaningful information past a point from which a process may diverge; accordingly, our semantic model takes the view that a process may perform *any* sequence of events after divergence. Thus the only reliable behaviours of a process are those in $T - D$.

Axiom 2 reflects the intuition that $\checkmark$ represents successful termination. In particular, there is no way a process may diverge after a $\checkmark$ unless it is already divergent.

Given a process P, its denotation $[\![P]\!]_{\mathcal{T}^\Downarrow} = (\mathsf{traces}_\perp(P), \mathsf{divergences}(P)) \in \mathcal{T}^\Downarrow$ is calculated by induction on the structure of P; in other words, the model $\mathcal{T}^\Downarrow$ is compositional. The complete list of clauses can be found in [Ros98, Chap. 8], and moreover the traces and divergences of a process may also be extracted from the operational semantics in straightforward fashion. We provide the inductive rules in Figures 4.7 and 4.8 to facilitate the proofs. In the last three rules in Figure 4.8, r ranges over $\Sigma^{*\checkmark}$, in accordance with Axiom 4. We recall that the lifting of the renaming relation R to traces is carried out pointwise. The definition of $s \underset{A}{\|} t$ in the rules for parallel composition was presented earlier in Figure 2.7 [Ros98].

Definition 4.4.1. *A process P is **livelock-free** if* $\mathsf{divergences}(P) = \emptyset$.

$$
\begin{aligned}
\mathsf{traces}(STOP) &= \{\langle\rangle\}\\
\mathsf{traces}(SKIP) &= \{\langle\rangle, \langle\checkmark\rangle\}\\
\mathsf{traces}(DIV) &= \{\langle\rangle\}\\
\mathsf{traces}(a \longrightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle^\frown t \mid t \in \mathsf{traces}(P)\}\\
\mathsf{traces}(P \mathrel{\Box} Q) &= \mathsf{traces}(P) \cup \mathsf{traces}(Q)\\
\mathsf{traces}(P \sqcap Q) &= \mathsf{traces}(P) \cup \mathsf{traces}(Q)\\
\mathsf{traces}(P \mathbin{;} Q) &= (\mathsf{traces}(P) \cap \Sigma^*) \cup \{t^\frown s \mid t^\frown\langle\checkmark\rangle \in \mathsf{traces}(P), s \in \mathsf{traces}(Q)\}\\
\mathsf{traces}(P \setminus A) &= \{t \upharpoonright (\Sigma \setminus A) \mid t \in \mathsf{traces}(P)\}\\
\mathsf{traces}(P[\![R]\!]) &= \{t \mid \exists\, s \in \mathsf{traces}(P) \bullet s \mathrel{R} t\}\\
\mathsf{traces}(P \underset{A}{\parallel} Q) &= \bigcup\{s \underset{A}{\parallel} t \mid s \in \mathsf{traces}(P), t \in \mathsf{traces}(Q)\}
\end{aligned}
$$

Figure 4.7: The model $\mathcal{T}^{\Downarrow}$: inductive rules for calculating traces

$$
\begin{aligned}
\mathsf{divergences}(STOP) &= \emptyset\\
\mathsf{divergences}(SKIP) &= \emptyset\\
\mathsf{divergences}(DIV) &= \Sigma^{*\checkmark}\\
\mathsf{divergences}(a \longrightarrow P) &= \{\langle a\rangle^\frown t \mid t \in \mathsf{divergences}(P)\}\\
\mathsf{divergences}(P \mathrel{\Box} Q) &= \mathsf{divergences}(P) \cup \mathsf{divergences}(Q)\\
\mathsf{divergences}(P \sqcap Q) &= \mathsf{divergences}(P) \cup \mathsf{divergences}(Q)\\
\mathsf{divergences}(P \mathbin{;} Q) &= \mathsf{divergences}(P) \cup \{t^\frown s \mid t^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(P), s \in \mathsf{divergences}(Q)\}\\
\mathsf{divergences}(P \setminus A) &= \{(t \upharpoonright (\Sigma \setminus A))^\frown r \mid t \in \mathsf{divergences}(P)\} \cup\\
&\quad\ \{(u \upharpoonright (\Sigma \setminus A))^\frown r \mid u \in \Sigma^\omega, u \upharpoonright (\Sigma \setminus A) \text{ finite}, \forall\, t < u \bullet t \in \mathsf{traces}_\perp(P)\}\\
\mathsf{divergences}(P[\![R]\!]) &= \{t^\frown r \mid \exists\, s \in \mathsf{divergences}(P) \cap \Sigma^* \bullet s \mathrel{R} t\}\\
\mathsf{divergences}(P \underset{A}{\parallel} Q) &= \{u^\frown r \mid \exists\, s \in \mathsf{traces}_\perp(P), \exists\, t \in \mathsf{traces}_\perp(Q) \bullet u \in (s \underset{A}{\parallel} t \cap \Sigma^*),\\
&\quad\ (s \in \mathsf{divergences}(P) \text{ or } t \in \mathsf{divergences}(Q))\}
\end{aligned}
$$

Figure 4.8: The model $\mathcal{T}^{\Downarrow}$: inductive rules for calculating divergences

4.4.2.2 Reasoning about Infinite Traces

In general, reasoning about livelock requires reasoning about infinite behaviours. Hiding a set of events $A \subseteq \Sigma$ from a process P introduces divergence if P is capable of performing an infinite unbroken sequence of events from A. Although our model only records the finite traces of a process, the finite-branching nature of our operators[4] entails (via König's lemma) that a process may perform an infinite trace u if and only if it can perform all finite prefixes of u. In other words, the set of finite traces of a process conveys enough information for deducing the set of its infinite traces as well. To keep the notation simple, given an infinite word $u \in \Sigma^\omega$, we will write

$$u \in \mathsf{traces}^\omega(P) \text{ whenever } \{t \in \Sigma^* \mid t < u\} \subseteq \mathsf{traces}(P),$$

where $<$ denotes the strong prefix order on $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Furthermore, we will write $\mathsf{traces}^\infty(P)$ to denote $\mathsf{traces}(P) \cup \mathsf{traces}^\omega(P)$, the set of all finite and infinite traces of P. We note that traces in $\mathsf{traces}^\omega(P)$, and hence finite prefixes thereof, cannot contain a $\checkmark$, which denotes successful termination.

We now state the semantic properties that we use in case of infinite traces. The proofs of the lemmas stated below can be found in Appendix A.1.

Lemma 4.4.2. *Let $u \in \mathsf{traces}^\omega(a \longrightarrow P)$. Then there exists $u' \in \mathsf{traces}^\omega(P)$, such that $u = \langle a \rangle ^\frown u'$.*

Lemma 4.4.3. *Let $u \in \mathsf{traces}^\omega(P \oplus Q)$ for $\oplus \in \{\Box, \sqcap\}$. Then $u \in \mathsf{traces}^\omega(P)$ or $u \in \mathsf{traces}^\omega(Q)$.*

Lemma 4.4.4. *Let $u \in \mathsf{traces}^\omega(P \mathbin{;} Q)$. Then $u \in \mathsf{traces}^\omega(P)$, or $u = t^\frown u'$ with $t^\frown\langle\checkmark\rangle \in \mathsf{traces}(P)$ and $u' \in \mathsf{traces}^\omega(Q)$.*

Lemma 4.4.5. *Let $u \in \mathsf{traces}^\omega(P \setminus A)$ and $P \setminus A$ be livelock-free. Then there exists $v \in \mathsf{traces}^\omega(P)$, such that $u = v \upharpoonright (\Sigma \backslash A)$.*

Lemma 4.4.6. *Let $u \in \mathsf{traces}^\omega(P[\![R]\!])$. Then there exists $v \in \mathsf{traces}^\omega(P)$, such that $v \mathrel{R} u$.*

Lemma 4.4.7. *Let $u \in \mathsf{traces}^\omega(P \underset{A}{\parallel} Q)$. Then there exist $u_1 \in \mathsf{traces}^\infty(P)$ and $u_2 \in \mathsf{traces}^\infty(Q)$, such that $u \in u_1 \underset{A}{\parallel} u_2$, and $u_1 \in \Sigma^\omega$ or $u_2 \in \Sigma^\omega$.*

[4]All CSP operators are finitely branching under the assumption that the alphabet Σ is finite and there is no unbounded nondeterminism [Ros98].

4.4.2.3 Handling Recursion

In this section we use standard notions from lattice theory which we presented in Section 2.1.3.1.

We interpret recursive processes in the standard way by introducing a partial order $\sqsubseteq$ on $\mathcal{T}^{\Downarrow}$. We write $(T_1, D_1) \sqsubseteq (T_2, D_2)$ if $T_2 \subseteq T_1$ and $D_2 \subseteq D_1$. In other words, the order on $\mathcal{T}^{\Downarrow}$ is reverse inclusion on both the trace and the divergence components. The resulting partial order $(\mathcal{T}^{\Downarrow}, \sqsubseteq)$ is a complete lattice. The bottom element of $(\mathcal{T}^{\Downarrow}, \sqsubseteq)$ is $(\Sigma^{*\checkmark}, \Sigma^{*\checkmark})$, i.e., the denotation of the immediately divergent process DIV. The top element is $(\{\langle\rangle\}, \emptyset)$, i.e., the denotation of the immediately deadlocking process $STOP$. The least upper bound and the greatest lower bound of a family $\{(T_i, D_i) \mid i \in I\}$ are given by $\bigsqcup_{i\in I}(T_i, D_i) = (\bigcap_{i\in I} T_i, \bigcap_{i\in I} D_i)$ and $\bigsqcap_{i\in I}(T_i, D_i) = (\bigcup_{i\in I} T_i, \bigcup_{i\in I} D_i)$, respectively.

It is readily verified that each n-ary CSP operator other than recursion can be interpreted as a Scott-continuous function $(\mathcal{T}^{\Downarrow})^n \to \mathcal{T}^{\Downarrow}$. The continuity of hiding rests on our assumption that Σ is finite (cf. [Ros98, Lemma 8.3.5]). By induction we have that any CSP expression P in variables $X_1, \ldots, X_n$ is interpreted as a Scott-continuous map $(\mathcal{T}^{\Downarrow})^n \to \mathcal{T}^{\Downarrow}$. Recursion is then interpreted using the least fixed-point operator $\mathrm{fix} : [\mathcal{T}^{\Downarrow} \to \mathcal{T}^{\Downarrow}] \to \mathcal{T}^{\Downarrow}$. For instance, $[\![\mu X \centerdot X]\!]_{\mathcal{T}^{\Downarrow}}$ is the least fixed point of the identity function on $\mathcal{T}^{\Downarrow}$, i.e., the immediately divergent process. Our analysis of livelock freedom, however, is based around an alternative treatment of fixed points in terms of metric spaces.

4.5 Processes and Metric Spaces

4.5.1 Metric Spaces

In this section, we recall standard definitions and facts concerning metric spaces [Sut75].

A *metric space* is a pair (A, d), where A is a non-empty set and $d : A \times A \to \mathbb{R}^+$ is a function satisfying the following three laws for all $x, y, z \in A$:

$$\begin{array}{rcll} d(x,y) = 0 & \Leftrightarrow & x = y & \text{diagonal law} \\ d(x,y) & = & d(y,x) & \text{symmetry law} \\ d(x,y) & \leq & d(x,z) + d(z,y) & \text{triangle inequality} \end{array}$$

We refer to A as *space* and to d as a *metric* on A. For a given space A, there may be multiple metrics on A. For example, the function $d : A \times A \to \{0, 1\}$ defined by:

$$d(x,y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

satisfies the three metric laws and is called the *discrete metric* on A.

A metric space (A, d) is an *ultrametric space* if the metric function f further satisfies the following inequality for all $x, y, z \in A$:

$$d(x,y) \ \leq \ \max(d(x,z), d(z,y)) \ \text{ ultrametric inequality}$$

For example, the discrete metric on A is also an ultrametric. Among other properties, an ultrametric guarantees that every triangle is isosceles.

Convergence and Closed Sets. Let (A, d) be a metric space and $s = \langle x_n \mid n \in \mathbb{N} \rangle$ be an infinite sequence of points in A. We say that s *converges* to a point $y \in A$ if, for each $\varepsilon > 0$, there exists $N_\varepsilon \in \mathbb{N}$, such that for every $n \geq N_\varepsilon$, $d(x_n, y) < \varepsilon$. s is a *Cauchy sequence* if, for each $\varepsilon > 0$, there exists $N_\varepsilon \in \mathbb{N}$, such that for every $n, m \geq N_\varepsilon$, $d(x_n, x_m) < \varepsilon$. In a metric space, every convergent sequence is a Cauchy sequence. However, the converse does not necessarily hold. Hence, the metric space (A, d) is a *complete metric space* if every Cauchy sequence converges. A given subset C of A is *closed* if whenever $\langle x_n \mid n \in \mathbb{N} \rangle$ is a sequence of points in C that converges to a point $y \in A$, then $y \in C$.

Functions and Fixed Points. Let (A, d) be a metric space and $f : A \to A$ be a self map on A. A point $a^* \in A$ is a *fixed point* of f if $f(a^*) = a^*$. We say that f is a *contraction (contractive map)* if there exists a non-negative constant $c < 1$ such that, for any $x, y \in A$,

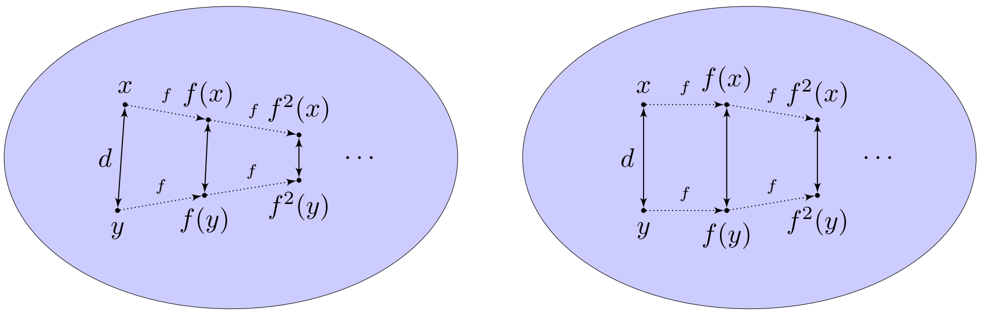
$$d(f(x), f(y)) \leqslant c \cdot d(x, y).$$

Intuitively, if f is a contraction, then the distance between any two point $x, y \in A$ is strictly greater (by some factor) than the distance between their images under f, as depicted in Figure 4.9(a). We say that f is *nonexpansive* if for any $x, y \in A$,

$$d(f(x), f(y)) \leqslant d(x, y).$$

Hence, a nonexpansive function f provides the guarantee that the distance between any two points $x, y \in A$ does not increase after an application of f, as illustrated in Figure 4.9(b).

In metric spaces, even if complete, nonexpansive functions may have many fixed points or none at all. Contractive functions, however, are guaranteed to have a fixed point and, furthermore, a *unique* fixed point.



(a) A contractive map (b) A nonexpansive map

Figure 4.9: Contractive and nonexpansive maps

Theorem 4.5.1 (Banach's Fixed Point Theorem)**.** *Let (A, d) be a complete metric space and $f : A \to A$ be a contraction on A with respect to the metric d. Then f has a unique fixed point. Furthermore, starting from* any *point in A, repeated application of f is guaranteed to converge to this unique fixed point.*[5]

4.5.2 Introducing a New Family of Metrics

Let $F(X)$ be a CSP term with a free variable X. Then F can be seen as a self map of $\mathcal{T}^{\Downarrow}$. Assume that there exists some metric on $\mathcal{T}^{\Downarrow}$ which is complete and under which F is a contraction. Then it follows from the Banach fixed point theorem [Sut75] (Theorem 4.5.1) that F has a unique (possibly divergent) fixed point $\mu X \bullet F(X)$ in $\mathcal{T}^{\Downarrow}$. Furthermore, starting from *any point* in $\mathcal{T}^{\Downarrow}$, for example *STOP*, iterated application of F is guaranteed to converge to this unique fixed point.

There may be several such metrics, or none at all. All we are interested in is to prove the existence of at least one. Fortunately, a class of suitable metrics can be systematically elicited from the sets of guards of a particular recursion.

[5]For every $x \in A$, the sequence $\langle f^n(x) \mid n \in \mathbb{N} \rangle$ is a Cauchy sequence, which is guaranteed to converge, owing to the completeness of the metric space.

Generally speaking, the metrics that we consider are all variants of the well-known 'longest common prefix' metric on traces, which were first studied by Roscoe in his doctoral dissertation [Ros82], and independently by de Bakker and Zucker [BZ82]. In this metric, the longer prefix two traces share, the closer they are, with the standard lifting to sets of traces and, therefore, to processes. Hence, the longer two processes exhibit the same behaviours, the shorter the distance between them. Formally, let Σ be a finite alphabet and let us consider the set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ of all finite and infinite words over Σ. In the setting of the 'longest common prefix' on traces, a metric function d on Σ^∞ would look like:

$$d(s,t) = \inf_{[0,1]}\{2^{-n} \mid s \text{ and } t \text{ possess a common prefix of length } n\},$$

and its extension to a metric function on the semantic domain $\mathcal{T}^\Downarrow$:

$$d(P,Q) = \inf_{[0,1]}\{2^{-n} \mid [\![P]\!]_{\mathcal{T}^\Downarrow} \text{ and } [\![Q]\!]_{\mathcal{T}^\Downarrow} \text{ are indistinguishable for traces up to length } n\}.$$

Example 4.5.2. *Let us consider the process $F(X) = a \longrightarrow X$. Then for any $s,t \in \Sigma^\infty$, $d(F(s),F(t)) = \frac{1}{2}d(s,t)$. Therefore, F is contractive with respect to d and hence $\mu X \centerdot F(X)$ has a unique fixed point $(\{\langle\rangle, \langle a\rangle, \langle a,a\rangle, \langle a,a,a\rangle, \ldots\}, \emptyset)$.*

The reason we need to consider variants of the 'longest common prefix' metric is that it fails to make the hiding operator nonexpansive:

Example 4.5.3. *Let us consider two processes $P = a \longrightarrow a \longrightarrow b \longrightarrow STOP$ and $Q = a \longrightarrow a \longrightarrow c \longrightarrow STOP$. Then $[\![P]\!]_{\mathcal{T}^\Downarrow} = (\{\langle\rangle, \langle a\rangle, \langle a,a\rangle, \langle a,a,b\rangle\}, \emptyset)$ and $[\![Q]\!]_{\mathcal{T}^\Downarrow} = (\{\langle\rangle, \langle a\rangle, \langle a,a\rangle, \langle a,a,c\rangle\}, \emptyset)$. Therefore $d(P,Q) = \frac{1}{4}$. However, $d(P \setminus \{a\}, Q \setminus \{a\}) = d((\{\langle\rangle, \langle b\rangle\}, \emptyset), (\{\langle\rangle, \langle c\rangle\}, \emptyset)) = 1$.*

The solution, in this particular case, is to change the definition of the length of a trace by only counting non-a events. We devise a new metric which is parametric in a given set of visible events $U \subseteq \Sigma$ and ensures that any CSP operator, other than recursion, is nonexpansive in each of its arguments. To formalise these ideas let us introduce a few auxiliary definitions. These are all parametric in a given set of events $U \subseteq \Sigma$.

Given a trace $s \in \Sigma^{*\checkmark}$, the U-length of s, denoted $\mathsf{length}_U(s)$, is defined to be the number of occurrences of events from U in s. Given a set of traces $T \subseteq \Sigma^{*\checkmark}$ and $n \in \mathbb{N}$, the restriction of T to U-length n is defined by:

$$T \upharpoonright_U n \mathrel{\widehat{=}} \{s \in T \mid \mathsf{length}_U(s) \leqslant n\}.$$

We extend this restriction operator to act on our semantic domain $\mathcal{T}^{\Downarrow}$ by defining $(T, D) \upharpoonright_U n \mathrel{\widehat{=}} (T', D')$, where

1. $D' = D \cup \{s^\frown t \mid s \in T \cap \Sigma^* \text{ and } \mathsf{length}_U(s) = n\}$.
2. $T' = D' \cup \{s \in T \mid \mathsf{length}_U(s) \leqslant n\}$.

Thus $P \upharpoonright_U n$ denotes a process which behaves like P until n events from the set U have occurred, after which it diverges unless it has already terminated. It is the least process which agrees with P on traces with U-length no greater than n.

We now define a metric d_U on $\mathcal{T}^{\Downarrow}$ as follows:

$$d_U(P, Q) \mathrel{\widehat{=}} \inf_{[0,1]}\{2^{-n} \mid P \upharpoonright_U n = Q \upharpoonright_U n\} \enspace .$$

Proposition 4.5.4. *Let $U \subseteq \Sigma$. Then $(\mathcal{T}^{\Downarrow}, d_U)$ is an ultrametric space.*

Proof. It is easy to prove that $(\mathcal{T}^{\Downarrow}, d_U)$ satisfies the following laws for every $P, Q, R \in \mathcal{T}^{\Downarrow}$:

$$\begin{array}{rcll} d_U(P,Q) = 0 & \Leftrightarrow & P =_{\mathcal{T}^{\Downarrow}} Q & \text{diagonal law} \\ d_U(P,Q) & = & d_U(Q,P) & \text{symmetry law} \\ d_U(P,Q) & \leq & d_U(P,R) + d_U(R,Q) & \text{triangle inequality} \\ d_U(P,Q) & \leq & \max(d_U(P,R), d_U(R,Q)) & \text{ultrametric inequality} \end{array}$$

The proofs for the first two laws are trivial. Regarding the triangle and ultrametric laws, let us suppose that $d_U(P, R) = 2^{-n}$, $d_U(R, Q) = 2^{-m}$ and $k = \min(n, m)$. Then, $P \upharpoonright_U k = R \upharpoonright_U k = Q \upharpoonright_U k$. Therefore,

$$d_u(P, Q) \leq 2^{-k} = \max(d_U(P, R), d_U(R, Q)) \leq d_U(P, R) + d_U(R, Q).$$

□

Notice that the function $U \mapsto d_U$ is antitone: if $U \subseteq V$, then $d_U \geqslant d_V$, i.e., for any $P, Q \in \mathcal{T}^{\Downarrow}$, $d_U(P, Q) \geq d_V(P, Q)$. In particular, the greatest of all the d_U is $d_\emptyset$; this is the discrete metric on $\mathcal{T}^{\Downarrow}$. Furthermore, the least of all the d_U is d_Σ; this is the standard 'longest common prefix'-style metric on $\mathcal{T}^{\Downarrow}$ as defined in [Ros98, Chap. 8].

Proposition 4.5.5 (Antitoneness). *If $U \subseteq V \subseteq \Sigma$, then for any $P, Q \in \mathcal{T}^{\Downarrow}$,*

$$d_U(P, Q) \geq d_V(P, Q).$$

Proposition 4.5.6. *Let $U \subseteq \Sigma$. Then $\mathcal{T}^{\Downarrow}$ equipped with the metric d_U is a complete ultrametric space and the set of livelock-free processes is a closed subset of $\mathcal{T}^{\Downarrow}$. Furthermore, if $F : \mathcal{T}^{\Downarrow} \to \mathcal{T}^{\Downarrow}$ is contractive with respect to d_U, then F has a unique fixed point given by $\lim_{n\to\infty} F^n(STOP)$. (Note that this fixed point may be divergent.)*

Proof. By Proposition 4.5.4, $(\mathcal{T}^{\Downarrow}, d_U)$ is an ultrametric space. The proofs that $(\mathcal{T}^{\Downarrow}, d_U)$ is a complete metric space and that set of livelock-free processes is a closed subset of $\mathcal{T}^{\Downarrow}$ are presented in Appendix A.2 (as Propositions A.2.2 and A.2.3, respectively).

Let $F : \mathcal{T}^{\Downarrow} \to \mathcal{T}^{\Downarrow}$ be contractive with respect to d_U. Since $(\mathcal{T}^{\Downarrow}, d_U)$ is a complete metric space, it follows from Banach's fixed point theorem (Theorem 4.5.1) that F has a unique fixed point given by $\lim_{n\to\infty} F^n(\theta)$, where θ can be any element of $\mathcal{T}^{\Downarrow}$ and, in particular, the process $STOP$. The unique fixed point may or may not be livelock free, however. $\square$

In the rest of this paper, the only metrics we are concerned with are those associated with some subset of Σ; accordingly, we freely identify metrics and sets when the context is unambiguous.

4.5.3 Nonexpansiveness of CSP Operators

Let us fix $U \subseteq \Sigma$. The following lemmas prove that each CSP operator, other than recursion, is at least nonexpansive with respect to d_U in each of its arguments (for some operators we need to impose certain conditions). The proofs of the lemmas can be found in Appendix A.3.

Lemma 4.5.7. *For any CSP processes P, P', Q and Q' the following inequalities hold:*

$$d_U(P \mathbin{\Box} Q, P' \mathbin{\Box} Q) \leq d_U(P, P') \text{ and } d_U(P \mathbin{\Box} Q, P \mathbin{\Box} Q') \leq d_U(Q, Q')$$

$$d_U(P \sqcap Q, P' \sqcap Q) \leq d_U(P, P') \text{ and } d_U(P \sqcap Q, P \sqcap Q') \leq d_U(Q, Q')$$

$$d_U(P \mathbin{⨾} Q, P' \mathbin{⨾} Q) \leq d_U(P, P') \text{ and } d_U(P \mathbin{⨾} Q, P \mathbin{⨾} Q') \leq d_U(Q, Q')$$

$$d_U(P \underset{A}{\parallel} Q, P' \underset{A}{\parallel} Q) \leq d_U(P, P') \text{ and } d_U(P \underset{A}{\parallel} Q, P \underset{A}{\parallel} Q') \leq d_U(Q, Q').$$

Lemma 4.5.8. *Let P and Q be CSP processes and let $a \in \Sigma$. Then:*

$$d_U(a \longrightarrow P, a \longrightarrow Q) \leq d_U(P, Q).$$

Furthermore, if $a \in U$, then the inequality is strict.

Lemma 4.5.9. *Let P and Q be CSP processes and let $A \subseteq \Sigma$ satisfy $A \cap U = \emptyset$. Then:*

$$d_U(P \setminus A, Q \setminus A) \leq d_U(P, Q).$$

Lemma 4.5.10. *Let P and Q be CSP processes, $R \subseteq \Sigma \times \Sigma$ be a renaming relation on Σ and $R(U) = \{y \mid \exists\, x \in U \bullet x\ R\ y\}$. Then:*

$$d_{R(U)}(P[\![R]\!], Q[\![R]\!]) \leq d_U(P, Q).$$

Lemma 4.5.11. *Let P, Q and Q' be CSP processes. Let P always communicate an event from U before it does a $\checkmark$. Then:*

$$d_U(P \mathbin{;} Q, P \mathbin{;} Q') \leq \frac{1}{2} d_U(Q, Q').$$

4.6 Static Livelock Analysis: General Framework

We present an algorithm based on a static analysis which conservatively flags processes that may livelock. In other words, any process classified as livelock-free really is livelock-free, although the converse may not hold.

Divergent behaviours originate in three different ways, two of which are non-trivial. The first is through direct use of the process *DIV*; the second comes from unguarded recursions; and the third is through hiding an event, or a set of events, which the process can perform infinitely often to the exclusion of all others.

Roscoe [Ros98, Chap. 8] addresses the second and third points by requiring that all recursions be *guarded*, i.e., always perform some event prior to recursing, and by banning use of the hiding operator under recursion. Our idea is to extend Roscoe's requirement that recursions should be guarded by stipulating that one may never hide *all* the guards. In addition, one may not hide a set of events which a process is able to perform infinitely often to the exclusion of all others. This will therefore involve a certain amount of book-keeping.

4.6.1 Nonexpansiveness and Guardedness

We first treat the issue of guardedness of the recursions. Our task is complicated by the renaming operator, in that a purported guard may become hidden only after several unwindings of a recursion. The following example illustrates some of the ways in which a recursion may fail to be guarded and thus diverge.

Example 4.6.1. *Let $\Sigma = \{a, b, a_0, a_1, \ldots, a_n\}$ and let $R = \{(a_i, a_{i+1}) \mid 0 \leqslant i < n\}$ and $S = \{(a, b), (b, a)\}$ be renaming relations on Σ. Consider the following processes.*

1. $\mu X \centerdot X$.
2. $\mu X \centerdot a \longrightarrow (X \setminus a)$.
3. $\mu X \centerdot (a \longrightarrow (X \setminus b)) \sqcap (b \longrightarrow (X \setminus a))$.
4. $\mu X \centerdot (a_0 \longrightarrow (X \setminus a_n)) \sqcap (a_0 \longrightarrow X[\![R]\!])$.
5. $\mu X \centerdot SKIP \sqcap a \longrightarrow (X \mathbin{;} (X[\![S]\!] \setminus b))$.

The first recursion is trivially unguarded. In the second recursion the guard a is hidden after the first recursive call. In the third process the guard in each summand is hidden in the other summand; this process will also diverge once it has performed a single event. In the fourth example we cannot choose a set of guards which is both stable under the renaming operator and does not contain a_n. This process, call it P, makes the following sequence of visible transitions:

$$P \xrightarrow{a_0} P \setminus a_n \xrightarrow{a_0} P[\![R]\!] \setminus a_n \xrightarrow{a_1} P[\![R]\!][\![R]\!] \setminus a_n \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} P[\![R]\!][\![R]\!] \ldots [\![R]\!] \setminus a_n.$$

But the last process diverges, since P can make an infinite sequence of a_0-transitions which get renamed to a_n by successive applications of R and are then hidden at the outermost level.

A cursory glance at the last process might suggest that it is guarded in $\{a\}$. However, similarly to the previous example, hiding and renaming conspire to produce divergent behaviour. In fact the process, call it P, can make an a-transition to $P \mathbin{;} (P[\![S]\!] \setminus b)$, and consequently to $(P[\![S]\!] \setminus b)[\![S]\!] \setminus b$ via two τ-transitions. But this last process can diverge. □

Given a variable X and a CSP term $P = P(X)$, we aim to define inductively a collection $\mathsf{C}_X(P)$ of metrics for which P is contractive as a function of X (bearing in mind that processes may have several free variables). It turns out that it is first necessary to identify those metrics in which P is merely nonexpansive as a function of X, the collection of which we denote $\mathsf{N}_X(P)$. Intuitively, the role of $\mathsf{N}_X(P)$ is to keep track of all hiding and renaming in P. A set $U \subseteq \Sigma$ then induces a metric d_U under which P is contractive in X provided P is nonexpansive in U and $\mu X \bullet P$ always communicates an event from U prior to recursing.

The intuitions underlying our definitions of nonexpansiveness and guardedness are the following [OPRW13]. By definition, P is contractive with respect to d_U (with contraction factor 1/2) if, for every $T_1, T_2 \in \mathcal{T}^{\Downarrow}$, it is the case that

$$d_U(P(T_1), P(T_2)) \leq \frac{1}{2} d_U(T_1, T_2)\,. \tag{4.1}$$

However, if P happens to apply a one-to-one renaming operator R to its argument, say, then it becomes necessary to rephrase (4.1) above as

$$d_V(P(T_1), P(T_2)) \leq \frac{1}{2} d_U(T_1, T_2)\,, \tag{4.2}$$

where d_V is a new metric such that $R(U) = V$. Indeed, since P renames events in U to ones in V, the distance between $P(T_1)$ and $P(T_2)$ must be measured with respect to the renamed events, rather than the original ones.

This leads us to the concept of a function that is contractive *with respect to two different metrics d_U and d_V*, in which the first metric is used to measure the distance between two inputs, whereas the second metric measures the distance between the corresponding two outputs of the function under consideration—see Figure 4.10. Following our convention of identifying sets and metrics, we would say that P is *contractive in the pair* (U, V).

This reasoning needs to be slightly refined in order to handle non-injective renamings as well as hiding. Our goal is then to define, by induction on the structure of CSP terms, a function $\mathsf{C}_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$, which associates to each CSP term $P(X)$ a set of pairs of metrics (U, V) such that (4.2) holds.

As pointed out above, in order to define such a function C_X, it is first necessary to compute a function $\mathsf{N}_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ which calculates, for every CSP term $P(X)$, a set of pairs of metrics (U, V) such that P is nonexpansive in (U, V), following the same convention of measuring the distance between inputs via the metric d_U and the distance between outputs via the metric d_V.

It is also necessary to calculate an auxiliary function $\mathsf{G} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$, which itself depends on a certain function $\mathsf{F} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$. This may seem problematic, since F itself depends on C_X, but this mutual recursion is well-defined because uses of F in the definition of G only occur on subterms, and likewise for uses of G in C_X and uses of C_X in F.

We provide the intuitions underlying the definitions of G and F later on, as these functions are introduced. For now let us finally remark that all the functions that we define are *conservative underapproximations*, i.e., sound, but not necessarily complete. For example, $\mathsf{N}_X(P)$ as defined below generates some but not necessarily all of the pairs of metrics that witness the nonexpansiveness of P.

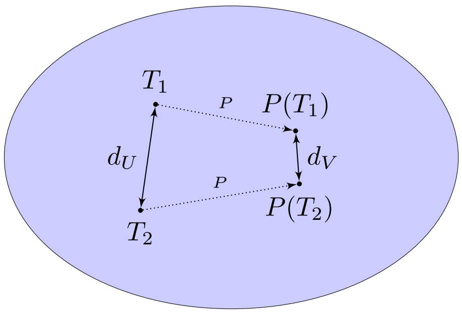


Figure 4.10: P is contractive in (U, V), i.e., when the distance between inputs is measured with respect to d_U and the distance between outputs is measured with respect to d_V.

The key property of the function N_X is given by the following proposition.

Proposition 4.6.2. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a CSP term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $\mathsf{N}_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.11. If $(U, V) \in N_X(P)$, then for all $T_1, T_2, \Theta_1, \ldots, \Theta_n \in \mathcal{T}^{\Downarrow}$, $d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) \leq d_U(T_1, T_2)$.*

Proof. The proof proceeds by structural induction on P and is presented in Appendix A.4. □

$$\mathsf{N}_X(P) \mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \quad \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:}$$

$$\begin{aligned}
\mathsf{N}_X(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{N}_X(P) \\
\mathsf{N}_X(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{N}_X(P_1) \cap \mathsf{N}_X(P_2) \ \text{ if } \oplus \in \{\sqcap, \Box, \fatsemi, \underset{A}{\parallel}\} \\
\mathsf{N}_X(P \setminus A) &\mathrel{\widehat{=}} \{(U,V) \mid (U,V') \in \mathsf{N}_X(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\} \\
\mathsf{N}_X(P[\![R]\!]) &\mathrel{\widehat{=}} \{(U,V) \mid (U,V') \in \mathsf{N}_X(P) \wedge R(V') \subseteq V\} \\
\mathsf{N}_X(X) &\mathrel{\widehat{=}} \{(U,V) \mid U \subseteq V\} \\
\mathsf{N}_X(\mu Y \centerdot P) &\mathrel{\widehat{=}} \{(U,V) \mid (U',V') \in \mathsf{N}_X(P) \wedge (V',V') \in \mathsf{N}_Y(P) \wedge U \subseteq U' \wedge V' \subseteq V\} \\
&\qquad \text{if } Y \neq X \ .
\end{aligned}$$

Figure 4.11: Nonexpansive sets

Let us note that, by construction, $\mathsf{N}_X(P)$ is always downwards-closed in its first component and upwards-closed in its second component, which is sound due to antitoneness (see Proposition 4.5.5).

Before defining $\mathsf{C}_X(P)$, we need an auxiliary construct denoted $\mathsf{G}(P)$. Intuitively, $\mathsf{G}(P) \subseteq \mathcal{P}(\Sigma)$ lists the 'guards' of ✓ for P. Formally:

Proposition 4.6.3. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $\mathsf{G} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.12. If $V \in \mathsf{G}(P)$, then, with any processes—and in particular DIV—substituted for the free variables of P, P must communicate an event from V before it can do a ✓.*

Proof. The proof proceeds by structural induction on P and is presented together with Proposition A.4.1 in Appendix A.4. □

Note that the inductive clauses for G that are given in Figure 4.12 make use of the collection of *fair sets* $\mathsf{F}(P_i)$ of P_i, which is presented later on. The definition is nonetheless well-founded since F is here only applied to subterms. The salient feature of $\mathsf{F}(P_i) \neq \emptyset$ is that the process P_i is guaranteed to be livelock-free.

We are now ready to define $C_X(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, whose central property is given by the following proposition.

$$
\begin{aligned}
\mathsf{G}(STOP) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma)\\
\mathsf{G}(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{G}(P) \cup \{V \mid a \in V\}\\
\mathsf{G}(SKIP) &\mathrel{\widehat{=}} \emptyset\\
\mathsf{G}(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{G}(P_1) \cap \mathsf{G}(P_2) \quad \text{if } \oplus \in \{\Box, \sqcap\}\\
\mathsf{G}(P_1 \operatorname{\text{;}} P_2) &\mathrel{\widehat{=}} \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if } P_1 \text{ is closed and } \mathsf{F}(P_1) \neq \emptyset\\ \mathsf{G}(P_1) & \text{otherwise} \end{cases}\\
\mathsf{G}(P_1 \underset{A}{\parallel} P_2) &\mathrel{\widehat{=}} \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if, for } i = 1,2,\ P_i \text{ is closed and } \mathsf{F}(P_i) \neq \emptyset\\ \mathsf{G}(P_1) \cap \mathsf{G}(P_2) & \text{otherwise} \end{cases}\\
\mathsf{G}(P \setminus A) &\mathrel{\widehat{=}} \begin{cases} \{V \mid V' \in \mathsf{G}(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\} & \text{if } P \text{ is closed and}\\ & (\emptyset, \Sigma - A) \in \mathsf{F}(P)\\ \emptyset & \text{otherwise} \end{cases}\\
\mathsf{G}(P[\![R]\!]) &\mathrel{\widehat{=}} \{V \mid V' \in \mathsf{G}(P) \wedge R(V') \subseteq V\}\\
\mathsf{G}(X) &\mathrel{\widehat{=}} \emptyset\\
\mathsf{G}(\mu X \bullet P) &\mathrel{\widehat{=}} \mathsf{G}(P) \ .
\end{aligned}
$$

Figure 4.12: Guard sets

Proposition 4.6.4. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $\mathsf{C}_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.13. If $(U, V) \in \mathsf{C}_X(P)$, then for all $T_1, T_2, \theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, $d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta})) \leq \frac{1}{2} d_U(T_1, T_2)$.*

Proof. The proof proceeds by structural induction on P and is presented together with Proposition A.4.1 in Appendix A.4 □

Note that contraction guarantees a unique fixed point, albeit not necessarily a livelock-free one. For instance, $P(X) = (a \longrightarrow X \setminus b) \Box (\mu Y \bullet b \longrightarrow Y)$ is contractive in (U, U) for any $U \subseteq \Sigma$ with $a \in U$ and $b \notin U$. Therefore, $P(X)$ has a unique fixed point, but this unique fixed point can diverge after a single event.

Let us also note that, similarly to Proposition 4.6.2, by construction, $\mathsf{C}_X(P)$ is always downwards-closed in its first component and upwards-closed in its second component, which is sound due to antitoneness (see Proposition 4.5.5).

$$\begin{aligned}
&\mathsf{C}_X(P) \mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \ \ \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:}\\
\mathsf{C}_X(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{C}_X(P) \cup \{(U,V) \in \mathsf{N}_X(P) \mid a \in V\}\\
\mathsf{C}_X(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{C}_X(P_1) \cap \mathsf{C}_X(P_2) \ \text{ if } \oplus \in \{\Box, \sqcap, \underset{A}{\|}\}\\
\mathsf{C}_X(P_1 \fatsemi P_2) &\mathrel{\widehat{=}} \mathsf{C}_X(P_1) \cap (\mathsf{C}_X(P_2) \cup \{(U,V) \in \mathsf{N}_X(P_2) \mid V \in \mathsf{G}(P_1)\})\\
\mathsf{C}_X(P \setminus A) &\mathrel{\widehat{=}} \{(U,V) \mid (U,V') \in \mathsf{C}_X(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\}\\
\mathsf{C}_X(P[\![R]\!]) &\mathrel{\widehat{=}} \{(U,V) \mid (U,V') \in \mathsf{C}_X(P) \wedge R(V') \subseteq V\}\\
\mathsf{C}_X(X) &\mathrel{\widehat{=}} \emptyset\\
\mathsf{C}_X(\mu Y \centerdot P) &\mathrel{\widehat{=}} \{(U,V) \mid (U',V') \in \mathsf{C}_X(P) \wedge (V',V') \in \mathsf{N}_Y(P) \wedge U \subseteq U' \wedge V' \subseteq V\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } Y \neq X \ .
\end{aligned}$$

Figure 4.13: Contractive sets

4.6.2 Fair Sets and Hiding

In order to prevent livelock, we must ensure that, whenever a process can perform an infinite[6] unbroken sequence of events from a particular set A, then we never hide the whole of A. To this end, we now associate to each CSP term P a collection of pairs of *fair sets* $\mathsf{F}(P) \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$: intuitively, this allows us to keep track of the events which the process is guaranteed to perform infinitely often in any infinite execution of P. As with nonexpansiveness and contractiveness, the potential presence of renaming and hiding requires us separately to keep track of events performed by the input processes and the output (or compound) process.

Definition 4.6.5. *Given a set $W \subseteq \Sigma$, we say that a process is W-*fair *if any of its infinite traces contains infinitely many events from W.*

Proposition 4.6.6. *Let $P(X_1, \ldots, X_n) = P(\overline{X})$ be a CSP term whose free variables are contained within the set $\{X_1, \ldots, X_n\}$. Let $\mathsf{F} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.14. If $(U,V) \in \mathsf{F}(P)$, then, for any collection of livelock-free, U-fair processes $\theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, the process $P(\theta_1, \ldots, \theta_n)$ is livelock-free and V-fair.*

[6]Recall our understanding that a process can 'perform' an infinite trace iff it can perform all its finite prefixes.

$$
\begin{aligned}
\mathsf{F}(\mathit{STOP}) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)\\
\mathsf{F}(a \longrightarrow P) &\mathrel{\widehat{=}} \mathsf{F}(P)\\
\mathsf{F}(\mathit{SKIP}) &\mathrel{\widehat{=}} \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)\\
\mathsf{F}(P_1 \oplus P_2) &\mathrel{\widehat{=}} \mathsf{F}(P_1) \cap \mathsf{F}(P_2) \ \text{ if } \oplus \in \{\sqcap, \Box, \fatsemi\}\\
\mathsf{F}(P_1 \underset{A}{\parallel} P_2) &\mathrel{\widehat{=}} (\mathsf{F}(P_1) \cap \mathsf{F}(P_2)) \cup\\
&\qquad \{(U_1 \cap U_2, V_1) \mid (U_1, V_1) \in \mathsf{F}(P_1) \wedge (U_2, A) \in \mathsf{F}(P_2)\} \cup\\
&\qquad \{(U_1 \cap U_2, V_2) \mid (U_2, V_2) \in \mathsf{F}(P_2) \wedge (U_1, A) \in \mathsf{F}(P_1)\}\\
\mathsf{F}(P \setminus A) &\mathrel{\widehat{=}} \{(U, V) \mid (U, V') \in \mathsf{F}(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\}\\
\mathsf{F}(P[\![R]\!]) &\mathrel{\widehat{=}} \{(U, V) \mid (U, V') \in \mathsf{F}(P) \wedge R(V') \subseteq V\}\\
\mathsf{F}(X) &\mathrel{\widehat{=}} \{(U, V) \mid U \subseteq V\}\\
\mathsf{F}(\mu X \centerdot P) &\mathrel{\widehat{=}} \begin{cases} \{(U, V) \mid (W, W) \in \mathsf{C}_X(P) \cap \mathsf{F}(P) \wedge U \subseteq W \subseteq V\} & \text{if } \mu X \centerdot P \text{ is open}\\ \mathcal{P}(\Sigma) \times \{V \mid (W, W) \in \mathsf{C}_X(P) \cap \mathsf{F}(P) \wedge W \subseteq V\} & \text{otherwise .} \end{cases}
\end{aligned}
$$

Figure 4.14: Fair sets

Proof. The proof proceeds by structural induction on P and is presented together with Proposition A.4.1 in Appendix A.4. □

Note that, by construction, $\mathsf{F}(P)$ is always downwards-closed in its first component and upwards-closed in its second component. This is sound since if $U \subseteq U'$ and P is U-fair, then P is automatically U'-fair as well.

We now obtain one of our main results as an immediate corollary:

Theorem 4.6.7. *Let P be a CSP process (i.e., a closed CSP term) not containing DIV in its syntax. If $\mathsf{F}(P) \neq \emptyset$, then P is livelock-free.*

Proof. Let $\mathsf{F}(P) \neq \emptyset$ and $(U, V) \in \mathsf{F}(P)$ for some $U, V \subseteq \Sigma$. Since P is closed, P has no free variables. Then, from Proposition 4.6.6, P is livelock-free and, furthermore, V-fair. □

Theorem 4.6.7 gives rise to a procedure for establishing livelock freedom of a given process P over an alphabet Σ, whose complexity is at most quadratic in the syntactic size of P and exponential in the cardinality of Σ: indeed, for a fixed Σ, we compute $\mathsf{N}_X(Q)$, $\mathsf{G}(Q)$, $\mathsf{C}_X(Q)$ and $F(Q)$ for every variable X appearing in P and every subterm Q of P. Since the number of variables and the number of subterms are both at most linear in the size of P, the computation is at most quadratic in P [OPRW13]. On the other hand, each of

$\mathsf{N}_X(Q)$, $\mathsf{C}_X(Q)$ and $F(Q)$ is a collection of pairs of subsets of Σ, whereas $\mathsf{G}(Q)$ is a collection of subsets of Σ. Thus for Σ not fixed, these pieces of data are potentially exponentially large.

In practice, applications often make use of moderately large alphabets, making the direct set-based approach described above prohibitively expensive. However, an inspection of the rules defining $\mathsf{N}_X(Q)$, $\mathsf{G}(Q)$, $\mathsf{C}_X(Q)$ and $F(Q)$ reveals that these objects can be represented *symbolically*, either as propositional formulas or as BDDs—further implementation details are provided in Section 4.8. As a result, the problem of deciding whether $\mathsf{F}(P) \neq \emptyset$ can be seen to lie in NP [OPRW13].

4.6.3 Examples and Discussion

Example 4.6.8. *Let us consider the process B^+ [Ros98] (see Figure 4.15) which implements a buffer of infinite size. B^+ is an infinite-state process and is therefore beyond the grasp of the explicit checker FDR. A first glance at the CSP encoding might lead us to mistakenly think that B^+ is divergent: if we look at the parallel composition, on the left-hand side we rename right to mid, on the right-hand side we rename left to mid, and then on top level we hide the event mid. However, B^+ turns out to be livelock-free—a formal (and not completely trivial) proof of livelock freedom is presented in [Ros98]. In contrast, our static analyser establishes livelock freedom automatically and instantaneously by affirming that B^+ is fair in any set $U \subseteq \Sigma$ with left $\in U$.*

$$
\begin{aligned}
&M = \{mid\}\\
&RtoM = [right \leftarrow mid, mid \leftarrow right]\\
&LtoM = [left \leftarrow mid, mid \leftarrow left]\\
&\\
&COPY = left \longrightarrow right \longrightarrow COPY\\
&B^+ = left \longrightarrow (B^+[\![RtoM]\!] \underset{\{mid\}}{\parallel} (right \longrightarrow COPY)[\![LtoM]\!]) \setminus \{mid\}
\end{aligned}
$$

Figure 4.15: An infinite buffer

Example 4.6.9. *Let us consider the abstracted version of the alternating bit protocol (ABP), whose script we recall below.*

$$\boxed{\begin{array}{l} \mathit{Send} = \mathit{in} \longrightarrow \mathit{Medium} \\ \mathit{Medium} = \mathit{out} \longrightarrow \mathit{Send} \mathrel{\Box} \mathit{error} \longrightarrow \mathit{Medium} \\ \\ \mathit{Fair} = \mathit{out} \longrightarrow \mathit{Fair} \mathrel{\Box} \mathit{error} \longrightarrow \mathit{out} \longrightarrow \mathit{Fair} \\ \\ \mathit{Network} = \mathit{Send} \underset{\{\mathit{error},\mathit{out}\}}{\parallel} \mathit{Fair} \\ \\ \mathit{System} = \mathit{Network} \setminus \{\mathit{error}\} \end{array}}$$

The expanded transition systems of the processes Send and Fair are depicted in Figure 4.16

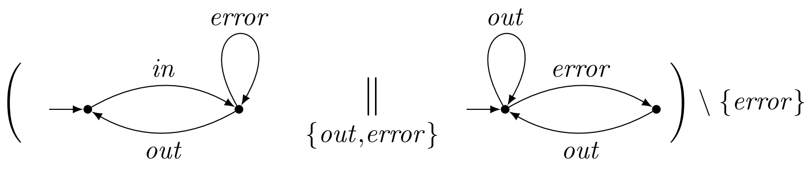


Figure 4.16: Abstracted ABP: transition systems

Using the systems of rules we presented, we calculate the sets of fair sets of Send, Fair, Network and System as follows (where the operator $\uparrow$ *denotes upper closure on* $\mathcal{P}(\mathcal{P}(\Sigma))$ *and* $\Sigma = \{\mathit{in}, \mathit{out}, \mathit{error}\}$*):*

$$\begin{aligned} \mathsf{F}(\mathit{Send}) &\mathrel{\widehat{=}} \uparrow \{\{\mathit{in}, \mathit{error}\}, \{\mathit{out}, \mathit{error}\}\} \\ \mathsf{F}(\mathit{Fair}) &\mathrel{\widehat{=}} \uparrow \{\{\mathit{out}\}\} \\ \mathsf{F}(\mathit{Network}) &\mathrel{\widehat{=}} \uparrow \{\{\mathit{out}\}, \{\mathit{in}, \mathit{error}\}\} \\ \mathsf{F}(\mathit{System}) &\mathrel{\widehat{=}} \uparrow \{\{\mathit{out}\}\} \end{aligned}$$

Therefore, System is livelock-free and any infinite trace of System contains infinitely many occurrences of the event out.

An interesting weakness of our framework is that it fails to establish the fact that System is also $\{\mathit{in}\}$*-fair. Indeed, since System is equivalent to the process* $B_1 = \mathit{in} \longrightarrow \mathit{out} \longrightarrow B_1$*, any infinite trace of System should also contain infinitely many occurrences of in. Therefore, the process* $\mathit{Network} \setminus \{\mathit{error}, \mathit{out}\}$*, which is equivalent to the process* $\mathit{IN} = \mathit{in} \longrightarrow \mathit{IN}$*, is livelock-free and* $\{\mathit{in}\}$*-fair. However,* $\mathsf{F}(\mathit{Network} \setminus \{\mathit{error}, \mathit{out}\}) = \emptyset$ *(thanks to the* F *rule* *for hiding) and therefore our framework would mark Network* $\setminus \{error, out\}$ *as potentially divergent.*

As the last example illustrates, the system of rules we developed in this section can sometimes be overly conservative. More specifically, our experiments indicated that, for a particular class of communication protocols, e.g., ABP and the sliding window protocol, where synchronisation plays a subtle but vital role, our analysis was inadequate.

4.7 Static Livelock Analysis for Structurally Finite-State Processes

The techniques developed in Sections 4.5.2 and 4.6 allow us to handle the widest range of CSP processes; among others, they enable one to establish livelock freedom of numerous infinite-state processes including examples making use of infinite buffers or unbounded counters, as illustrated in Example 4.6.8. Such processes are of course beyond the reach of explicit-state model checkers such as FDR. In order to create them in CSP, it is necessary to use devices such as recursing under the parallel operator. In practice, however, the vast majority of processes tend to be finite state.

Let us therefore define a CSP process to be *structurally finite state* if it never syntactically recurses under any of parallel, the left-hand side of a sequential composition, hiding or renaming.

More precisely, we first define a notion of *sequential* CSP terms: $STOP$, $SKIP$, and X are sequential; if P and Q are sequential, then so are $a \longrightarrow P$, $P \sqcap Q$, $P \mathrel{\Box} Q$ and $\mu X \centerdot P$; and if in addition P is closed, then $P \mathbin{⨾} Q$, $P \setminus A$ and $P[\![R]\!]$ are sequential. Observe that sequential processes (i.e., closed sequential terms) give rise to labelled transition systems of size linear in the length of their syntax.

Now any closed sequential term is deemed to be structurally finite state; and if P and Q are structurally finite state, then so are $a \longrightarrow P$, $P \sqcap Q$, $P \mathrel{\Box} Q$, $P \underset{A}{\parallel} Q$, $P \mathbin{⨾} Q$, $P \setminus A$ and $P[\![R]\!]$. Note that structurally finite-state CSP terms are always closed, i.e., are processes.

We write SEQ and $\overline{\mathsf{SFS}}$ to denote the collections of sequential CSP terms and structurally finite-state CSP processes, respectively. We refer to the minimal closed SEQ terms (i.e., minimal SEQ processes) as *atomic* $\overline{\mathsf{SFS}}$ processes. As their transition systems are linear in the size of their syntax, they are relatively cheap to expand and compute explicitly. Compound $\overline{\mathsf{SFS}}$ processes can be obtained from simpler counterparts by using any CSP operators other than recursion.

Whether a given process is structurally finite state can easily be established by syntactic inspection, for example by using Bekič's theorem [Win93] (see Section 4.3) and analysing the resulting μ expression. For such processes, it turns out that we can substantially both simplify and sharpen our livelock analysis. More precisely, the computation of nonexpansive and contractive data is circumvented by instead directly examining closed sequential components in isolation. For those we compute exact data; we start becoming conservative and losing precision only in the compositional rules for handling compound processes, as depicted in Figure 4.17. Furthermore, the absence of free variables in compound processes makes some of the earlier fairness calculations unnecessary, thereby allowing more elaborate and finer data to be computed efficiently, as we now explain.

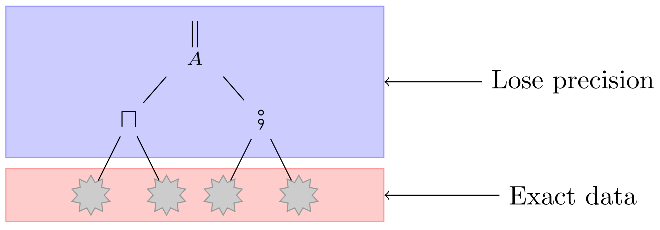


Figure 4.17: Precision layers

Definition 4.7.1. *Let u be an infinite trace over Σ and let $F, C \subseteq \Sigma$ be two sets of events. We say that:*

- *u is* fair in F *if, for each $a \in F$, u contains infinitely many occurrences of a,*[7]
- *u is* co-fair in C *if, for each $b \in C$, u contains only finitely many occurrences of b.*

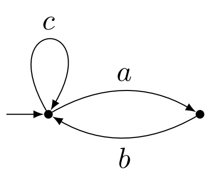
As an example, let us consider the trace $u = b(ac)^\omega$. It will be true if we state that u is fair in $F = \{a\}$ and co-fair in $C = \{b\}$. However, if we want to be more precise and compute exact data, we need to say that u is fair in $F = \{a, c\}$ and co-fair in $C = \{b\}$. In

[7]Note that this notion of 'fairness' differs from that used in the previous section.

any case, we can think of the pair (F, C) as an abstraction (overapproximation) of u—it abstracts u into the set of all infinite traces that are fair in F and co-fair in C. In the latter case, the pair $(F, C) = (\{a, c\}, \{b\})$ provides an exact overapproximation.

We lift this to sets of traces in the following way: let $T \subseteq \Sigma^\omega$ be a set of infinite traces over Σ, and let $\mathcal{F} = \{(F_1, C_1), \ldots, (F_k, C_k)\} \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ be a collection of pairs of subsets of Σ. We say that T is *fair/co-fair in* $\mathcal{F}$ provided that, for every infinite trace $u \in T$, there exists a pair $(F_i, C_i) \in \mathcal{F}$ such that u is both fair in F_i and co-fair in C_i.

To generalise the concept to an LTS representing all possible executions of a process P, we need to capture this information for all infinite traces of P. In other words, taking an arbitrary infinite trace u of P, we need to investigate all possible alternatives for u. As an example, let us assume that P has the following transition system (where $\Sigma = \{a, b, c\}$):



There are three alternatives for u, depending on which simple cycles get visited infinitely often and which only finitely often:

1. $u \in ((ab) + c)^*(ab)^\omega$. In this case, u would be fair in $\{a, b\}$ and co-fair in $\{c\}$.

2. $u \in ((ab) + c)^*(c)^\omega$. In this case, u would be fair in $\{c\}$ and co-fair in $\{a, b\}$.

3. $u \in ((ab) + c)^*((ab)^+(c)^+)^\omega$. This case corresponds to the scenario when u visits both simple cycles infinitely often and implies that u is fair in $\{a, b, c\}$ and co-fair in $\emptyset$.

Hence P is fair/co-fair in the following collection of pairs of sets of events:

$$\mathcal{F} = \{\ (\{a, b\}, \{c\}), \quad (\{c\}, \{a, b\}), \quad (\{a, b, c\}, \emptyset)\ \}.$$

4.7.1 Goal

Our ultimate goal is the following: given a structurally finite-state process P, we aim to compute:

- a collection of *fair/co-fair* pairs of disjoint sets $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\} \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, together with

- a Boolean-valued *livelock flag* $\delta(P) \in \{\text{true}, \text{false}\}$, such that,

if $\delta(P) = \text{false}$, then P is livelock-free and, moreover, $\mathsf{traces}^{\omega}(P)$ is fair/co-fair in $\Phi(P)$.

4.7.2 Handling Atomic $\overline{\mathsf{SFS}}$ Processes

For P an atomic $\overline{\mathsf{SFS}}$ process (i.e., a minimal sequential process), let us denote by M_P its associated labelled transition system as derived from the operational semantics, and let us assume that we construct M_P so that all states are reachable from the initial state. As noted earlier, M_P has size linear in the syntactic description of P. We then compute the livelock flag $\delta(P)$ and the set of fair/co-fair pairs $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\}$ exactly, directly from M_P.

More precisely, we set $\delta(P)$ to true or false depending on whether or not P can eventually diverge, i.e., whether or not M_P contains a τ-cycle. In order to determine this, we use Tarjan's algorithm [CLRS01], which employs a nested depth-first search and is linear in the number of states in M_P.

If the livelock flag $\delta(P)$ is false, we compute the set of fair/co-fair pairs $\Phi(P)$. We want to include the pair of disjoint sets of events (F, C) to $\Phi(P)$ if and only if the transition system of P exhibits an infinite trace which is fair in F and co-fair in C (where $F \neq \emptyset$). Note that this will guarantee that our abstraction will be exact. Let us also observe that if P has no infinite traces, $\Phi(P)$ will be empty.

It is worth pointing out how this can be achieved efficiently. Let Σ_P be the alphabet of P. Given a non-empty set $L \subseteq \Sigma_P$ of events, we delete all $(\Sigma - L)$-labelled transitions from P's labelled transition system. If the resulting graph contains a (not necessarily reachable) strongly connected component which comprises *every single* event in L, we include $(L, \Sigma - L)$ as a fair/co-fair pair for P, and otherwise we do not.

Of course, in actual implementations, it is not necessary to explicitly iterate over all possible subsets of Σ. The computation we described can be carried out symbolically using a Boolean circuit of size polynomial in P, using well-known circuit algorithms for computing the transitive closure of relations. Consequently, $\Phi(P)$ can be represented symbolically and compactly either as a BDD or a propositional formula. Further implementation details are provided in Section 4.8.

4.7.2.1 Optimisations Considering Strongly Connected Components

For optimisation reasons, following the ideas behind divide-and-conquer algorithms, we actually first compute the SCC quotient graph of P (which is a directed acyclic graph) and obtain a set of reachable strongly connected edge-labelled directed graphs $\{P_1, \ldots, P_n\}$, as we shall now explain.

An *edge-labelled directed graph* is simply a labelled transition system without an initial state, i.e., a triple $\langle S, A, T\rangle$, where S is a finite set of vertices (or states), A is a finite set of labels and $R \subseteq S \times A \times S$ is a transition relation.

Let $M_P = \langle S, \mathsf{init}, \Sigma^{\tau\checkmark}, T\rangle$ be the labelled transition system associated with P. M_P induces a directed graph $G_P = (S, E)$, where $E = \{s \longrightarrow s' \mid \exists\, a \in \Sigma^{\tau\checkmark} \centerdot s \xrightarrow{a} s'\}$. We compute the SCC quotient graph $G_P^{\mathrm{SCC}} = (S^{\mathrm{SCC}}, E^{\mathrm{SCC}})$ of G_P using Tarjan's algorithm [CLRS01], which is linear in $|\, S\,|$. The set of vertices $S^{\mathrm{SCC}} = \{S_1, \ldots, S_n\}$ is a partition of the state space S of P and each vertex S_i forms a (maximal) strongly connected component (SCC) in G_P. We then let P_i be the edge-labelled directed graph induced from M_P by the set of states S_i. In other words, $P_i = \langle S_i, A_i, T_i\rangle$, where $T_i = \{s \xrightarrow{a}_i s' \mid s, s' \in S_i, \exists\, a \in \Sigma^{\tau\checkmark} \centerdot s \xrightarrow{a} s'\}$ and $A_i = \{a \in \Sigma^{\tau\checkmark} \mid \exists\, s, s' \in S_i \centerdot s \xrightarrow{a}_i s'\}$.

We note that, because G_P^{SCC} is a directed acyclic graph, every infinite path of P eventually ends up in precisely one SCC S_i, in which it stays forever. Since we are interested in computing fair and co-fair sets of events for infinite traces, it is sufficient therefore to consider the set of edge-labelled directed graphs $\{P_1, \ldots, P_n\}$ in isolation. For each P_i we compute the livelock flag $\delta(P_i)$ and the set of fair/co-fair pairs $\Phi(P_i)$, and then we let:

$$\delta(P) = \bigvee_{1 \le i \le n} \delta(P_i), \qquad \Phi(P) = \bigcup_{1 \le i \le n} \Phi(P_i).$$

4.7.3 Compositional Rules for Compound $\overline{\mathsf{SFS}}$ Processes

Proposition 4.7.2. *Let P be a structurally finite-state process. Let $\Phi : \overline{\mathsf{SFS}} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ and $\delta : \overline{\mathsf{SFS}} \longrightarrow \{\text{true}, \text{false}\}$ be defined recursively on the structure of P as shown in Figures 4.18 and 4.19, respectively. Then, if $\delta(P) = \text{false}$, P is livelock-free. Moreover, if $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\}$, then, for each infinite trace u of P, there exists $1 \le i \le k$, such that u is fair in F_i and u is co-fair in C_i.*

$$\begin{aligned}
\Phi(P) &\mathrel{\widehat{=}} \text{computed from } P\text{'s LTS (see Section 4.7.2)} \\
&\qquad \textbf{whenever } P \textbf{ is an atomic } \overline{\mathsf{SFS}} \textbf{ process; otherwise:} \\
\Phi(a \longrightarrow P) &\mathrel{\widehat{=}} \Phi(P) \\
\Phi(P_1 \oplus P_2) &\mathrel{\widehat{=}} \Phi(P_1) \cup \Phi(P_2) \quad \text{if } \oplus \in \{\sqcap, \Box, {}_{9}^{\circ}\} \\
\Phi(P_1 \underset{A}{\parallel} P_2) &\mathrel{\widehat{=}} \{(F, C) \mid F \cap C = \emptyset \wedge (F_i, C_i) \in \Phi(P_i) \text{ for } i = 1, 2 \wedge \\
&\qquad\qquad F = F_1 \cup F_2 \wedge \\
&\qquad\qquad C = (C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A))\} \cup \\
&\quad \{(F, C) \mid (F, C) \in \Phi(P_1) \wedge F \cap A = \emptyset\} \cup \\
&\quad \{(F, C) \mid (F, C) \in \Phi(P_2) \wedge F \cap A = \emptyset\} \\
\Phi(P \setminus A) &\mathrel{\widehat{=}} \{(F - A, C \cup A) \mid (F, C) \in \Phi(P)\} \\
\Phi(P[\![R]\!]) &\mathrel{\widehat{=}} \{(F, C) \mid (F', C') \in \Phi(P) \wedge F' \subseteq R^{-1}(F) \wedge F \subseteq R(F') \wedge \\
&\qquad\qquad C = \{b \in \Sigma \mid R^{-1}(b) \subseteq C'\}\} \ .
\end{aligned}$$

Figure 4.18: Fair/co-fair sets

$$\begin{aligned}
\delta(P) &\mathrel{\widehat{=}} \text{computed from } P\text{'s LTS (see Section 4.7.2)} \\
&\qquad \textbf{whenever } P \textbf{ is an atomic } \overline{\mathsf{SFS}} \textbf{ process; otherwise:} \\
\delta(a \longrightarrow P) &\mathrel{\widehat{=}} \delta(P) \\
\delta(P_1 \oplus P_2) &\mathrel{\widehat{=}} \delta(P_1) \vee \delta(P_2) \quad \text{if } \oplus \in \{\sqcap, \Box, \underset{A}{\parallel}, {}_{9}^{\circ}\} \\
\delta(P \setminus A) &\mathrel{\widehat{=}} \begin{cases} \text{false} & \text{if } \delta(P) = \text{false and, for each } (F, C) \in \Phi(P),\ F - A \neq \emptyset \\ \text{true} & \text{otherwise} \end{cases} \\
\delta(P[\![R]\!]) &\mathrel{\widehat{=}} \delta(P) \ .
\end{aligned}$$

Figure 4.19: δ-bit

Proof. The proof proceeds by structural induction on P and is presented in Appendix A.5. □

Let us observe that by construction, all fair/co-fair pairs of sets thus generated remain disjoint; this is key in the rule for parallel composition, where the fair/co-fair data of individual subcomponents enables one to rule out certain pairs for the resulting parallel process. Also, as shown in the proof, whenever (F, P) appears as a fair/co-fair pair in some

$\Phi(P)$, F is never empty.

Let us also remark that the δ clause for the hiding operator is phrased here in a way that makes the rule as intuitively clear as possible. In practice, one however need not iterate over all possible pairs $(F, C) \in \Phi(P)$: it is simpler instead to evaluate the negation, an existential calculation which is easily integrated within either a SAT or BDD implementation. Further details are provided in Section 4.8.

4.7.4 Examples and Discussion

4.7.4.1 The Abstracted Version of the ABP

To illustrate the precision of the system of rules for $\overline{\mathsf{SFS}}$ processes, let us get back to the abstracted version of the alternating bit protocol and try to establish that the process $\mathit{System} = \mathit{Network} \setminus \{\mathit{error}, \mathit{out}\} = (\mathit{Send} \underset{\{\mathit{error},\mathit{out}\}}{\|} \mathit{Fair}) \setminus \{\mathit{error}, \mathit{out}\}$ is livelock-free. As we demonstrated in Section 4.6.3, the collections of rules defined for the general framework were not accurate enough to infer that.

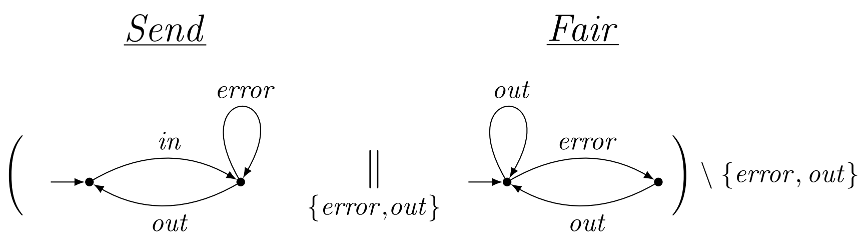


Figure 4.20: Abstracted version of the ABP

The processes *Send* and *Fair* depicted in Figure 4.20 are both atomic $\overline{\mathsf{SFS}}$ processes—for those we apply the algorithms described in Section 4.7.2 to conclude that $\delta(\mathit{Send}) = \delta(\mathit{Fair}) = \text{false}$ and, regarding the set of fair/co-fair pairs:

$$\Phi(\mathit{Send}) = \{\ (\{\mathit{in}, \mathit{out}\}, \{\mathit{error}\}), \quad (\{\mathit{error}\}, \{\mathit{in}, \mathit{out}\}), \quad (\{\mathit{error}, \mathit{in}, \mathit{out}\}, \emptyset)\ \},$$
$$\Phi(\mathit{Fair}) = \{\ (\{\mathit{out}\}, \{\mathit{in}, \mathit{error}\}), \quad (\{\mathit{error}, \mathit{out}\}, \{\mathit{in}\})\ \}.$$

Now let us consider the process $\mathit{Network} = \mathit{Send} \underset{\{\mathit{error},\mathit{out}\}}{\|} \mathit{Fair}$.

Since both *Send* and *Fair* are livelock-free, there is no way of having a divergence in *Network*, which is confirmed by the rule $\delta(\mathit{Network}) = \delta(\mathit{Send}) \vee \delta(\mathit{Fair}) = \text{false}$.

Let us now have a look at the Φ rule for parallel composition. Since none of the fair/co-fair pairs (F, C) of either *Send* or *Fair* satisfies $F \cap A = \emptyset$, where $A = \{\mathit{error}, \mathit{out}\}$ is the synchronisation set of the parallel composition, we can conclude the following:

1. We can only use the first set-comprehension clause for assembling the fair/co-fair pairs of *Network*.
2. *Both Send* and *Fair* contribute infinite traces to any infinite trace u of *Network*, i.e., $u = u_1 \underset{A}{\parallel} u_2$, where u_1 in $\mathsf{traces}^{\omega}(\mathit{Send})$ and $u_2 \in \mathsf{traces}^{\omega}(\mathit{Fair})$.

Intuitively, every infinite trace of *Fair*, and in particular u_2, contains infinitely many occurrences of *out*. Since *Send* and *Fair* synchronise on *out*, u_1 also contains infinitely many occurrences of *out*. But in u_1, *out* occurs infinitely often precisely whenever *in* occurs infinitely often. Therefore, u_1, and hence also u, both contain infinitely many occurrences of *in*. Therefore u is fair in *in*.

Formally, since both u_1 and u_2 are infinite, we need to consider every pair $((F_1, C_1), (F_2, C_2))$ in the Cartesian product of $\Phi(\mathit{Send})$ and $\Phi(\mathit{Fair})$, decide whether to discard it and, if not, figure out how to merge appropriately the pair of pairs into a single pair (F, C).

One of the crucial observations is the following. For a in the synchronisation set $A = \{\mathit{error}, \mathit{out}\}$, the number of occurrences of a in u_1, u_2 and u should be the same. Therefore we can discard all those pairs $((F_1, C_1), (F_2, C_2))$ such that there is $a \in A$ with $a \in F_1 \cap C_2$ or $a \in C_1 \cap F_2$. This leaves us with only two pairs:

1. $((\{\mathit{in}, \mathit{out}\}, \{\mathit{error}\}), (\{\mathit{out}\}, \{\mathit{in}, \mathit{error}\}))$, and
2. $((\{\mathit{error}, \mathit{in}, \mathit{out}\}, \emptyset), (\{\mathit{error}, \mathit{out}\}, \{\mathit{in}\}))$.

The important question now is what do we do with the event *in* which does not belong to the synchronisation set A. The reasoning we apply is that u is fair in *in* if at least one of u_1 and u_2 is fair in *in*, and u is co-fair in *in* if both u_1 and u_2 are co-fair in *in*. Then from the first pair we obtain $(F, C) = (\{\mathit{in}, \mathit{out}\}, \{\mathit{error}\})$ and from the second pair we obtain $(F, C) = (\{\mathit{error}, \mathit{in}, \mathit{out},\}, \emptyset)$. Hence we obtain the following final result for $\Phi(\mathit{Network})$, which confirms that every infinite trace of *Network* contains infinitely many occurrences of *in*:

$$\Phi(\mathit{Network}) = \{\ (\{\mathit{in}, \mathit{out}\}, \{\mathit{error}\}), \quad (\{\mathit{error}, \mathit{in}, \mathit{out}\}, \emptyset)\ \}$$

Now the only thing that remains is to handle the hiding operator, i.e., analyse $\mathit{System} = \mathit{Network} \setminus \{\mathit{error}, \mathit{out}\}$. Since for all $(F, C) \in \Phi(\mathit{Network})$, $F - \{\mathit{error}, \mathit{out}\} \neq \emptyset$, $\delta(\mathit{System})$ = false, i.e., we establish, as required, that *System* is livelock-free. As a nice consequence $\Phi(\mathit{System}) = \{(\{\mathit{in}\}, \{\mathit{error}, \mathit{out}\})\}$ asserts that every infinite trace u of *System* contains infinitely many occurrences of *in* and only finitely many occurrences of *out* and *error*. $\Box$

4.7.4.2 Static Livelock Analysis Algorithm

Theorems 4.6.7 and 4.7.2 yield a conservative algorithm for establishing livelock freedom: given a CSP process P (which we will assume does not contain DIV in its syntax), determine first whether P is structurally finite state. If so, assert that P is livelock-free if $\delta(P) =$ false, and otherwise report an inconclusive result. If P is not structurally finite state, assert that P is livelock-free if $\mathsf{F}(P) \neq \emptyset$, and otherwise report an inconclusive result.

The complexity of this procedure is in the worst case quadratic in the syntactic size of P and exponential in the cardinality of Σ, based on a similar line of reasoning as that presented following Theorem 4.6.7. Likewise, determining for P an $\overline{\mathsf{SFS}}$ process whether $\delta(P)$ is true is easily seen to lie in NP [OPRW13].

It is perhaps useful to illustrate how the inherent incompleteness of our procedure can manifest itself in very simple ways. For example, let $P = a \longrightarrow Q$ and $Q = (a \longrightarrow P) \,\Box\, (b \longrightarrow Q)$, and let $R = (P \underset{\{a,b\}}{\parallel} Q) \setminus b$ (see Figure 4.21). Using Bekič's procedure, R is readily seen to be a structurally finite-state process. Moreover, R is clearly livelock-free, yet $\delta(R) =$ true and $\mathsf{F}(R) = \emptyset$. Intuitively, establishing livelock freedom here requires some form of state-space exploration, to see that the 'divergent' state $(Q \underset{\{a,b\}}{\parallel} Q) \setminus b$ of R is in fact unreachable, but that is precisely the sort of reasoning that our static analysis algorithm is not geared to do.

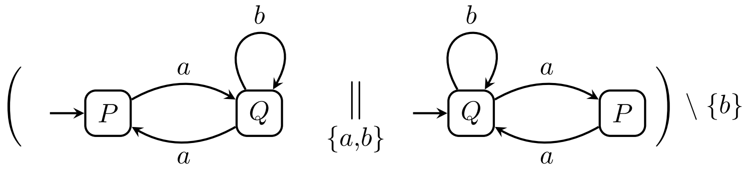


Figure 4.21: Static livelock analysis: a false positive

Nonetheless, we have found in practice that our approach succeeded in establishing livelock freedom for a wide range of existing benchmarks; we report on some of our experiments in Section 4.9.

Finally, it is worth noting that, for structurally finite-state processes, Theorem 4.7.2 is stronger than Theorem 4.6.7—it correctly classifies a larger class of processes as being livelock-free—and empirically has also been found to yield faster algorithms.

4.7.4.3 Notions of Fairness

It is perhaps interesting to compare how our notions of fairness and co-fairness differ from the ones defined in acceptance conditions in ω-automata, and more specifically, in Büchi, Müller, Rabin and Street automata.

A finite ω-automaton is a quintuple $M = \langle S, \mathsf{init}, \Sigma, T, \mathcal{F} \rangle$, where, similarly to transition systems, S is a finite set of states, init is the initial state, Σ is a finite alphabet and $T \subseteq S \times \Sigma \times S$ is a transition relation. $\mathcal{F}$ specifies the condition for accepting infinite words and is defined in different ways depending on the type of the automaton: $\mathcal{F} \subseteq S$ for Büchi automata, $\mathcal{F} \subseteq \mathcal{P}(\mathcal{P}(S))$ for Müller and generalised Büchi automata, and $\mathcal{F} \subseteq \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S))$ for Rabin and Street automata. In the following definitions, given an infinite sequence of states $\rho \in S^\omega$,

$$\inf(\rho) = \{s \in S \mid s \text{ occurs infinitely many times in } \rho\}.$$

Let $\alpha = \langle a_0, a_1, a_2, \ldots \rangle \in \Sigma^\omega$ be an infinite word over Σ. A run of M on the input word α is an infinite sequence of states $\rho = \langle s_0, s_1, s_2, \ldots \rangle$, such that $s_0 = \mathsf{init}$ and for all $i \geq 0$, $(s_i, a_i, s_{i+1}) \in T$. In Table 4.1 we specify the conditions necessary for ρ to be an accepting run of M on α. In all cases, the automaton M accepts α if there exists an accepting run ρ of M on α.

We note that in ω-automata acceptance conditions are defined on the set of states, whereas our notions of fairness and co-fairness target instead the set of events. Nevertheless, even if interpreted in the state setting, our concept is different from all of the above. It is most closely reminiscent of Rabin's acceptance condition; however, while Rabin stipulates $\inf(\rho) \cap F_i \neq \emptyset$, we would require $F_i \subseteq \inf(\rho)$.

Automaton	$\mathcal{F}$	ρ accepting on α if and only if:
Büchi	$\mathcal{F} \subseteq S$	$\inf(\rho) \cap \mathcal{F} \neq \emptyset$
Gen. Büchi	$\mathcal{F} = \{F_i \mid i \in I \ \wedge \ F_i \subseteq S\}$	$\forall F \in \mathcal{F}, \inf(\rho) \cap F \neq \emptyset$
Müller	$\mathcal{F} \subseteq \mathcal{P}(\mathcal{P}(S))$	$\inf(\rho) \in \mathcal{F}$
Rabin	$\mathcal{F} = \{(F_i, C_i) \mid i \in I \ \wedge \ F_i, C_i \subseteq S\}$	$\exists i \in I \centerdot \inf(\rho) \cap F_i \neq \emptyset \ \wedge \ \inf(\rho) \cap C_i = \emptyset$
Street	$\mathcal{F} = \{(F_i, C_i) \mid i \in I \ \wedge \ F_i, C_i \subseteq S\}$	$\forall i \in I \centerdot \inf(\rho) \cap F_i \neq \emptyset \Rightarrow \inf(\rho) \cap C_i \neq \emptyset$

Table 4.1: Acceptance conditions for ω-automata

4.8 Implementation Details and Symbolic Encoding

We have implemented both the general framework and the framework for structurally finite-state processes in a tool called SLAP, which is an acronym for Static Livelock Analyser of Processes. Computationally, the crux of our algorithms revolves around the generation and manipulation of sets. The algorithms fit very naturally into a symbolic paradigm; hence SLAP is fully symbolic. The choice of an underlying symbolic engine is configurable, with support for using a SAT engine (based on MiniSAT 2.0), a BDD engine (based on CUDD 2.4.2), or running a SAT and a BDD analyser in parallel and reporting the results of the first one to finish. SLAP is written in C++.

We have also integrated the framework for analysing structurally finite-state processes directly in FDR [AGL+12], where it now constitutes an alternative back-end for establishing livelock freedom. The binaries for the latter can be downloaded from the following location:

`http://www.cs.ox.ac.uk/projects/concurrency-tools/slap/`

In the remainder of this section we focus on the details regarding the symbolic part of our frameworks and algorithms.

In general, because we need to encode sets of sets of events, binary encoding is inadequate and we use one-hot Boolean encoding [KB05], i.e., for each $a \in \Sigma^{\tau\checkmark}$ we employ a Boolean variable which is also written a. The Boolean formula a then encodes all sets of events $\{A \subseteq \Sigma^{\tau\checkmark} \mid a \in A\}$ (see also Section 3.4.1). For the $\overline{\mathsf{SFS}}$ framework we use a single vector y of $\mid \Sigma^{\tau\checkmark} \mid$ Boolean variables, whereas for the general framework we employ two copies: one vector x for modelling the U component and another y for modelling the V

component (see Propositions 4.6.2, 4.6.4 and 4.6.6). In addition, we use auxiliary copies of variables for constructing more complex expressions using quantifiers and substitution. For those we use primed versions of x and y.

It is important to note that SAT techniques enable us to find a *single* fair or fair/co-fair set of a process. An advantage of this approach is the efficiency of modern SAT solvers. However, we need to introduce fresh vectors of variables for each instance of (even the same) subprocess. This is necessary because the SAT solver might need to generate different fair or fair/co-fair sets for a given term or a process, depending on the context in which it appears.

Using BDDs [Bry86] enables us to find *all possible* fair or fair/co-fair sets that the system of rules are capable of detecting. Hence, we do not need to duplicate subprocess encodings, however, we need to take care of variable orderings which can dramatically influence the size of the resulting BDD. We use variable ordering similar to the ones proposed in [Par02] and adopted by the probabilistic model checker PRISM [HKNP06, KNP11]. BDDs generally generate more compact representations compared to SAT encodings due to their canonicity and capability of capturing regularities and common patterns.

4.8.1 The $\overline{\mathsf{SFS}}$ Framework

4.8.1.1 Computing Fair/Co-Fair Sets for Atomic $\overline{\mathsf{SFS}}$ Processes

Let P be an atomic $\overline{\mathsf{SFS}}$ process and let us suppose that we have already established that $\delta(P) = \text{false}$, i.e., that P is livelock-free. As we described in Section 4.7.2, we next generate a collection of fair/co-fair pairs of disjoint sets $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\} \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ such that for every $1 \leq i \leq k$,

$$(F_i, C_i) \in \Phi(P) \longleftrightarrow \exists\, u \in \mathsf{traces}^{\omega}(P) \centerdot u \text{ is fair in } F_i \text{ and co-fair in } C_i. \tag{4.3}$$

The computation of $\Phi(P)$ is carried out directly on the labelled transition system M_P associated with P (in which unreachable states have been excised). Let us fix $M_P = \langle S, \mathsf{init}, \Sigma_P, \longrightarrow \rangle$ and let us suppose that P is a subcomponent of a system with alphabet Σ.

For a particular non-empty $L \subseteq \Sigma_P$, deciding whether or not to include $(L, \Sigma - L)$ to $\Phi(P)$ (lines 3 – 4) can be carried out in PTIME. More specifically, after obtaining G_L, we

Algorithm 5 Computing $\Phi(P)$

```
1: $\Phi(P) = \emptyset$
2: **for every** non-empty set $L \subseteq \Sigma_P$ **do**
3:     obtain a labelled graph $G_L$ from $P$'s LTS (having pruned unreachable states) by deleting
       all $(\Sigma - L)$-labelled transitions
4:     **if** $G_L$ contains a (not necessarily reachable) SCC which comprises *every* event in $L$ **then**
5:         include $(L, \Sigma - L)$ to $\Phi(P)$
6:     **end if**
7: **end for**
8: return $\Phi(P)$
```

can check whether there exists $s \in S$, such that for every $a \in L$, there exists a transition $src \xrightarrow{a} dest$, such that there are paths from s to src as well as from $dest$ back to s, as illustrated in Figure 4.22 for $L = \{in, out, error\}$. Note that such paths necessarily consist entirely of events in $L \cup \{\tau\}$.

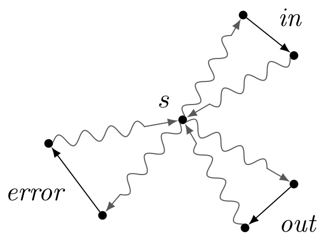


Figure 4.22: Calculating fair/co-fair sets for atomic $\overline{\mathsf{SFS}}$ processes

In fact, we can encode this symbolically for all possible subsets of Σ_P via the following Boolean formula:

$$MaxSCC = \bigvee_{s \in S} \Big\{ \bigwedge_{a \in \Sigma_P} \big[\neg a \ \vee \bigvee_{src \xrightarrow{a} dest} \big(\mathsf{Path}(s, src) \ \wedge \ \mathsf{Path}(dest, s)\big)\big] \Big\}, \tag{4.4}$$

where:

1. For all $s, t \in S$, $\mathsf{Path}(s, t)$ encodes all symbolic traces over Σ_P from s to t of length at most $|\,S\,|$, i.e., all symbolic traces of length at most the longest simple path in M_P. In order to compute $\mathsf{Path}(s, t)$ for all $s, t \in S$ simultaneously, we extend standard algorithms for computing the transitive closure of the adjacency matrix of the transition relation of M_P, such as Floyd-Warshall, iterative squaring or successive

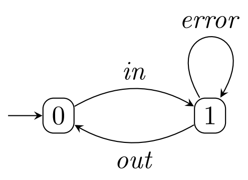
adjacency-matrix multiplications. Since the order of events on those traces is irrelevant to fairness and co-fairness, we do not employ symbolic state variables and use just a single copy of event variables to carry out the computation, as illustrated in Figure 4.23. We note that in those algorithms we do not check whether we reach a fixed point in the computation. As a consequence, if using a SAT encoding, the resulting formulas may contain redundancies.

2. The Boolean formula (4.4) contains an implicit iterator over all possible subsets L of $\Sigma_P \cup \{\tau\}$. In order to exclude the options of $L = \emptyset$ or $L = \{\tau\}$, we conjoin the formula with the restriction $\bigvee_{a \in \Sigma_P} a$.

3. We need to also declare all infinite traces of P as co-fair in $\Sigma - \Sigma_P$. To do so, we add another Boolean conjunct $\bigwedge_{a \in (\Sigma - \Sigma_P)} \neg a$.

The Boolean encoding of $\Phi(P)$ then looks as follows:

$$\Phi(P) = (\bigvee_{a \in \Sigma_P} a) \ \wedge \ (\bigwedge_{a \in (\Sigma - \Sigma_P)} \neg a) \ \wedge \ \mathit{MaxSCC}. \tag{4.5}$$



$$\mathsf{A} = \begin{bmatrix} \text{false} & \mathit{in} \\ \mathit{out} & \mathit{error} \end{bmatrix}$$

$$\mathsf{Path} = \begin{bmatrix} (\mathit{in} \wedge \mathit{out}) & (\mathit{in}) \vee (\mathit{in} \wedge \mathit{error}) \\ (\mathit{out}) \vee (\mathit{error} \wedge \mathit{out}) & (\mathit{error}) \vee (\mathit{out} \wedge \mathit{in}) \vee (\mathit{error} \wedge \mathit{error}) \end{bmatrix}$$

Figure 4.23: A symbolic representation of the adjacency matrix and the path matrix of the process *Send*. The path matrix is computed using successive matrix multiplications.

The Key: PTIME Algorithms and Circuits. As we have already established, given a process P and a non-empty set of events $L \subseteq \Sigma_P$, deciding whether or not to include $(L, \Sigma - L)$ to $\Phi(P)$ can be carried out in PTIME. Therefore, for the particular P and L, there exists a polynomial-size variable-free Boolean circuit that outputs true if and only if the pair $(L, \Sigma - L)$ is a fair/co-fair pair of P[8].

[8]This follows from the PTIME-hardness of circuit value.

Let us fix P and let us observe, moreover, that the construction of the variable-free circuit does not depend on the particular choice of L (see (4.4) and (4.5)). Therefore, we can leave the Σ_P input gates of the circuit as Boolean variables [Pap04]. What we obtain is a compact circuit of size polynomial in the syntax of P that encodes the computation of $\Phi(P)$ once and for all possible inputs, i.e., for all exponentially many subsets of Σ_P. We remark that the size of the circuit is polynomial in the size of P's LTS, which in turn is polynomial in P's syntactic description, since we are dealing with atomic, i.e., sequential, $\overline{\mathsf{SFS}}$ processes.

Since the circuit is of polynomial size, it can be turned into a polynomial-size (equisatisfiable) Boolean formula using, e.g., Tseitin encoding [Tse68] (see Section 2.1.4). The circuit can be also turned into a BDD, in which case the size of the BDD could potentially blow-up; however, in practice this is usually not the case. Consequently, the Boolean-formula or the BDD encoding of $\Phi(P)$ can be plugged into our compositional rules and be queried on demand when necessary, which fits very nicely into our symbolic framework.

4.8.1.2 Encoding Compositional Rules

The encodings of the rules for computing the livelock flag $\delta(P)$ and the collections of fair/co-fair pairs $\Phi(P)$ of a compound $\overline{\mathsf{SFS}}$ process P (see Theorem 4.7.2) are given in Figures 4.24 and 4.25 for Boolean formulas (i.e., for SAT) and in Figures 4.26 and 4.27 for BDDs.

4.8.2 The General Framework

The BDD and SAT encodings of the rules for computing the nonexpansive, guard, contractive and fair sets of CSP terms (see Theorems 4.6.2, 4.6.3, 4.6.4 and 4.6.6) are formalised similarly to the ones for the structurally finite-state processes. Just to give a flavour of the scheme and to illustrate the employment of two vectors of event variables, we provide the BDD encoding of the rules for computing nonexpansive sets in Figure 4.28.

In the encoding, the vectors of Boolean variables x and y model, respectively, the U and V components of the pairs of sets of events. To understand the meaning of the encoding operators $\mathsf{UClosure}, \mathsf{DClosure}$ and $\mathsf{UDClosure}$, suppose the formula $\varphi(x, y)$ encodes the set of pairs of sets of events $A = \{(U, V) \mid \text{some property}\}$ and the formula $\psi(y)$ encodes the set of sets of events $B = \{V \mid \text{some property}\}$. Then the formulas

$$
\begin{aligned}
\Phi(a \longrightarrow P)(y) &\mathrel{\widehat{=}} \Phi(P)(y) \\
\Phi(P_1 \oplus P_2)(y) &\mathrel{\widehat{=}} \Phi(P_1)(y') \ \wedge\ \Phi(P_2)(y'') \ \wedge\ \big[\bigwedge_{a\in\Sigma} a(y) \leftrightarrow a(y') \ \vee\ \bigwedge_{a\in\Sigma} a(y) \leftrightarrow a(y'')\big] \ \text{ if } \oplus \in \{\sqcap, \Box, \fatsemi\} \\
\Phi(P_1 \underset{A}{\|} P_2)(y) &\mathrel{\widehat{=}} \Phi(P_1)(y') \ \wedge\ \Phi(P_2)(y'') \ \wedge \\
&\qquad \big[\{\bigwedge_{a\in A} \neg a(y') \ \wedge\ \bigwedge_{a\in\Sigma} a(y) \leftrightarrow a(y')\} \ \vee \\
&\qquad \{\bigwedge_{a\in A} \neg a(y'') \ \wedge\ \bigwedge_{a\in\Sigma} a(y) \leftrightarrow a(y'')\} \ \vee \\
&\qquad \{\bigwedge_{a\in\Sigma} a(y) \leftrightarrow (a(y') \vee a(y'')) \ \wedge\ \bigwedge_{a\in A} \neg a(y) \leftrightarrow (\neg a(y') \vee \neg a(y'')) \\
&\qquad\qquad \wedge\ \bigwedge_{a\in\Sigma\setminus A} \neg a(y) \leftrightarrow (\neg a(y') \wedge \neg a(y''))\}\big] \\
\Phi(P \setminus A)(y) &\mathrel{\widehat{=}} \Phi(P)(y') \ \wedge\ \bigwedge_{a\in\Sigma\setminus A} a(y) \leftrightarrow a(y') \ \wedge\ \bigwedge_{a\in A} \neg a(y) \\
\Phi(P[\![R]\!])(y) &\mathrel{\widehat{=}} \Phi(P)(y') \ \wedge\ \bigwedge_{a\in\Sigma} [a(y') \rightarrow (\bigvee_{a\,R\,b} b(y))] \ \wedge\ \bigwedge_{b\in\Sigma} [(\bigwedge_{c\,R\,b} \neg c(y')) \rightarrow \neg b(y)]
\end{aligned}
$$

Figure 4.24: SAT encoding of $\Phi(P)$

$$
\delta(P \setminus A) \mathrel{\widehat{=}} \delta(P) \ \vee\ \Big(\neg\big[\Phi(P)(y) \rightarrow (\bigvee_{b\in\Sigma\setminus A} b(y))\big] \text{ is SAT}\Big)
$$

Figure 4.25: SAT encoding of $\delta(P)$

$\mathsf{UClosure}(A)(x,y), \mathsf{UDClosure}(A)(x,y), \mathsf{UClosure}(B)(y)$ and $\mathsf{DClosure}(B)(y)$ encode, respectively, the sets $\{(U,V) \mid (U,V') \in A \wedge V' \subseteq V\}$, $\{(U,V) \mid (U',V') \in A \wedge U \subseteq U' \wedge V' \subseteq V\}$, $\{V \mid V' \in B \wedge V' \subseteq V\}$ and $\{V \mid V' \in B \wedge V \subseteq V'\}$:

$$
\begin{aligned}
\mathsf{UClosure}(A)(x,y) &\mathrel{\widehat{=}} \exists\, y' \bullet \varphi(x,y') \wedge \bigwedge_i (y'_i \rightarrow y_i) \\
\mathsf{UDClosure}(A)(x,y) &\mathrel{\widehat{=}} \exists\, x'y' \bullet \varphi(x',y') \wedge \bigwedge_i (x_i \rightarrow x'_i) \wedge \bigwedge_i (y'_i \rightarrow y_i) \\
\mathsf{UClosure}(B)(y) &\mathrel{\widehat{=}} \exists\, y' \bullet \psi(y') \wedge \bigwedge_i (y'_i \rightarrow y_i) \\
\mathsf{DClosure}(B)(y) &\mathrel{\widehat{=}} \exists\, y' \bullet \psi(y') \wedge \bigwedge_i (y_i \rightarrow y'_i)
\end{aligned}
$$

$$
\begin{aligned}
\Phi(a \longrightarrow P)(y) &\mathrel{\widehat{=}} \Phi(P)(y) \\
\Phi(P_1 \oplus P_2)(y) &\mathrel{\widehat{=}} \Phi(P_1)(y) \ \wedge \ \Phi(P_2)(y) \ \text{ if } \oplus \in \{\sqcap, \Box, ⨟\} \\
\Phi(P_1 \underset{A}{\|} P_2)(y) &\mathrel{\widehat{=}} \exists\, y' \,\exists\, y'' \bullet \Phi(P_1)(y') \ \wedge \ \Phi(P_2)(y'') \ \wedge \\
&\qquad [\{ \bigwedge_{a \in A} \neg a(y') \ \wedge \ \bigwedge_{a \in \Sigma} a(y) \leftrightarrow a(y') \} \vee \\
&\qquad \{ \bigwedge_{a \in A} \neg a(y'') \ \wedge \ \bigwedge_{a \in \Sigma} a(y) \leftrightarrow a(y'') \} \vee \\
&\qquad \{ \bigwedge_{a \in \Sigma} a(y) \leftrightarrow (a(y') \vee a(y'')) \ \wedge \ \bigwedge_{a \in A} \neg a(y) \leftrightarrow (\neg a(y') \vee \neg a(y'')) \\
&\qquad\qquad \wedge \ \bigwedge_{a \in \Sigma \setminus A} \neg a(y) \leftrightarrow (\neg a(y') \wedge \neg a(y'')) \}] \\
\Phi(P \setminus A)(y) &\mathrel{\widehat{=}} \big[\exists\, y_A \bullet \Phi(P)(y) \big] \ \wedge \ \bigwedge_{a \in A} \neg a(y) \\
\Phi(P[\![R]\!])(y) &\mathrel{\widehat{=}} \exists\, y' \bullet \Phi(P)(y') \ \wedge \ \bigwedge_{a \in \Sigma} [a(y') \rightarrow (\bigvee_{a\,R\,b} b(y))] \ \wedge \ \bigwedge_{b \in \Sigma} [(\bigwedge_{c\,R\,b} \neg c(y')) \rightarrow \neg b(y)]
\end{aligned}
$$

Figure 4.26: BDD encoding of $\Phi(P)$

$$
\delta(P \setminus A) \mathrel{\widehat{=}} \delta(P) \ \vee \ \Big(\big[\Phi(P)(y) \rightarrow (\bigvee_{b \in \Sigma \setminus A} b(y)) \big] \text{ is not valid} \Big)
$$

Figure 4.27: BDD encoding of $\delta(P)$

4.9 Experimental Results

As we pointed out in Section 4.7, for structurally finite-state processes Theorem 4.7.2 is stronger than Theorem 4.6.7—it correctly classifies a larger class of processes as being livelock-free. Moreover, apart from improving on accuracy, the $\overline{\mathsf{SFS}}$ framework outperforms the general framework also in terms of efficiency. Therefore, given an input process P, SLAP first determines whether P is structurally finite-state. If so, it follows the $\overline{\mathsf{SFS}}$ framework and otherwise the general framework. Hence, in this section for all test cases using structurally finite-state processes we cite the results obtained using the $\overline{\mathsf{SFS}}$ framework.

All experiments were carried out on a 3.07GHz Intel Xeon processor running under Ubuntu with 8 GB of RAM. Times in seconds are given in Tables 4.2 and 4.3, with * indicating a 30-minute timeout and — denoting out of memory.

$$\begin{aligned}
\mathsf{N}_X(P)(x,y) &\mathrel{\widehat{=}} \text{true} \quad \textbf{whenever } X \textbf{ is not free in } P\textbf{; otherwise:}\\
\mathsf{N}_X(a \longrightarrow P)(x,y) &\mathrel{\widehat{=}} \mathsf{N}_X(P)(x,y)\\
\mathsf{N}_X(P_1 \oplus P_2)(x,y) &\mathrel{\widehat{=}} \mathsf{N}_X(P_1)(x,y) \ \wedge\ \mathsf{N}_X(P_2)(x,y) \quad \textbf{if } \oplus \in \{\sqcap, \Box, \mathbin{;}, \underset{A}{\parallel}\}\\
\mathsf{N}_X(P \setminus A)(x,y) &\mathrel{\widehat{=}} \mathsf{UClosure}(\mathsf{N}_X(P)(x,y) \ \wedge\ \chi(\{V \mid V \subseteq \Sigma - A\})(y))\\
&\mathrel{\widehat{=}} \mathsf{UClosure}(\mathsf{N}_X(P)(x,y) \ \wedge\ \mathsf{DClosure}(\Sigma - A)(y))\\
\mathsf{N}_X(P[\![R]\!])(x,y) &\mathrel{\widehat{=}} \mathsf{UClosure}(\exists\, y' \centerdot \mathsf{N}_X(P)(x,y') \ \wedge\ \rho(y',y))\\
\mathsf{N}_X(X)(x,y) &\mathrel{\widehat{=}} \bigwedge_i (x_i \rightarrow y_i)\\
\mathsf{N}_X(\mu Y \centerdot P)(x,y) &\mathrel{\widehat{=}} \mathsf{UDClosure}(\mathsf{N}_X(P)(x,y) \ \wedge\ \exists\, x' \centerdot (\mathsf{N}_Y(P)(x',y) \ \wedge\ \bigwedge_i (x'_i \leftrightarrow y_i))) \quad \textbf{if } Y \neq X
\end{aligned}$$

Figure 4.28: BDD encoding of $\mathsf{N}_X(P)$

4.9.1 Infinite-State Processes

While the state-of-the-art CSP refinement checker FDR [Ros94, G$^+$05] can only handle finite-state processes, SLAP is able to establish livelock freedom of the widest range of CSP process, including infinite-state ones.

We give two examples of infinite-state processes that fail to be $\overline{\mathsf{SFS}}$ and, while being beyond the capabilities of FDR, are handled easily by SLAP using the rules outlined in Section 4.6. An abstracted CSP script implementing an infinite-capacity buffer [Ros98] was presented as Example 4.6.8 in Section 4.6.3. The process B^+ is not $\overline{\mathsf{SFS}}$ due to the recursive call of B^+ under parallel composition. Two versions of an infinite counter [Ros98] are presented in Figure 4.29. They both fail to be $\overline{\mathsf{SFS}}$ due to the recursive calls under interleaving. The experimental results are presented in Table 4.2. The times are given in seconds.

$$\begin{aligned}
CNT &= up \longrightarrow (CNT \mathbin{|||} down \longrightarrow STOP)\\
CNT' &= up \longrightarrow (CNT' \mathbin{|||} down \longrightarrow CNT')
\end{aligned}$$

Figure 4.29: Infinite counters

Table 4.2: Static livelock analysis: experimental results for infinite-state processes. Times reported are in seconds.

Benchmark	**SLAP** (BDD)	**SLAP** (SAT)
B^+	0.02	0.02
CNT	0.01	0.01
CNT'	0.01	0

4.9.2 Structurally Finite-State Processes

We experimented with a wide range of benchmarks, including parametrised, parallelised, and piped version of Milner's scheduler, the alternating bit protocol (ABP), the sliding window protocol (SWP), the mad postman network routing protocol [YJ89], the classical dining philosophers, a distributed database algorithm, the time-dependent bully leader election algorithm and Fischer's mutual exclsuion protocol, etc.[9] In all our examples, internal communications were hidden, so that livelock freedom can be viewed as a progress or liveness property. All benchmarks were livelock-free, although the reader familiar with the above examples will be aware that manually establishing livelock freedom for several of these can be a subtle exercise (consider, e.g., the alternating bit protocol described in Section 4.2.1).

In all cases apart from the distributed database algorithm (for which the result was inconclusive), SLAP was indeed correctly able to assert livelock freedom (save for rare instances of timing out). Livelock freedom for the distributed database algorithm turns out to be remarkably complex; see [Ros98] for details and [Ros90] for a (40-page!) proof of correctness. In almost all instances, both the BDD and the SAT engines of SLAP substantially outperformed the state-of-the-art CSP model checker FDR, often completing orders of magnitude faster. On the whole, the BDD engine and the SAT engine performed comparably, with occasional discrepancies. In particular, the SAT engine ran quickly out of memory or timed out when dealing with time-dependent systems (where the discrete passage of time is modelled in CSP by using a special event called *tock*, see [Ros98, Ros11b]). Times in seconds are given in Table 4.3, with * indicating a 30-minute timeout and — denoting out of memory (before timing out).

[9]Scripts and descriptions for all benchmarks are available from the website associated with [Ros11b].

Table 4.3: Static livelock analysis: experimental results for $\overline{\mathsf{SFS}}$ processes. Times reported are in seconds, with * denoting a 30-minute timeout and — denoting out of memory.

Benchmark	**FDR**	**SLAP** (BDD)	**SLAP** (SAT)
Milner-10	0	0.06	0.05
Milner-15	0	0.19	0.14
Milner-20	409	0.63	0.28
Milner-21	948	0.73	0.23
Milner-22	*	0.93	0.25
Milner-25	*	1.63	0.41
Milner-30	*	7.56	0.8
ABP-0	0	0.03	0.11
ABP-0-inter-2	0	0.03	0.23
ABP-0-inter-3	23	0.06	0.35
ABP-0-inter-4	*	0.08	0.47
ABP-0-inter-5	*	0.09	0.63
ABP-0-pipe-2	0	0.04	0.35
ABP-0-pipe-3	2	0.06	0.75
ABP-0-pipe-4	175	0.08	1.27
ABP-0-pipe-5	*	0.10	1.85
ABP-0-pipe-6	*	0.11	2.91
ABP-4	0	0.11	*
ABP-4-inter-2	39	0.16	*
ABP-4-inter-3	*	0.22	*
ABP-4-inter-7	*	0.39	*
ABP-4-pipe-2	12	0.38	*
ABP-4-pipe-3	*	0.38	*
ABP-4-pipe-7	*	0.39	*
Bully-3	0	0	—
Bully-4	0	0	—
Bully-5	6	26	—
Bully-6	413	*	—
Bully-7	*	*	—

Benchmark	**FDR**	**SLAP** (BDD)	**SLAP** (SAT)
SWP-1	0	0.03	7.06
SWP-2	0	0.46	*
SWP-3	0	46.81	*
SWP-1-inter-2	0	0.04	14.84
SWP-1-inter-3	31	0.06	24.02
SWP-1-inter-4	*	0.08	29.44
SWP-1-inter-7	*	0.13	58.82
SWP-2-inter-2	170	0.71	*
SWP-2-inter-3	*	0.94	*
SWP-1-pipe-2	0	0.04	28.09
SWP-1-pipe-3	0	0.07	66.71
SWP-1-pipe-4	3	0.09	121.09
SWP-1-pipe-5	246	0.10	192.39
SWP-1-pipe-7	*	0.14	399.55
Philosophers-5	0	0.30	0.10
Philosophers-7	2	1.62	0.21
Philosophers-8	20	2.51	0.35
Philosophers-9	140	3.98	0.50
Philosophers-10	960	7.49	0.72
Mad Postman-2	0	0.06	0.03
Mad Postman-3	6	*	0.20
Mad Postman-4	*	*	0.89
Mad Postman-5	*	*	4.21
Mad Postman-6	*	*	20.75
Fischer-4	0	0	647
Fischer-5	1	0	*
Fischer-6	15	0	*
Fischer-7	213	3	*
Fischer-8	*	22	*

Details About the Test Cases. We give some details about the parameters that we vary to obtain multiple test instances with increasing size and difficulty. For Milner's scheduler and the dining philosophers we vary the number of cell processes and philosophers, respectively. In the latter, we hide on top all events modelling picking up and putting down forks—the resulting process gives the impression of potentially divergent, but in fact it is not. For this test case, we also established livelock freedom of the (deadlock-free) solutions using a butler or a single left-handed philosopher. The running times for all three versions of the dining philosophers are more or less the same. For the bully leader election algorithm, we vary the number of processors participating in the protocol and fix the three timeout values T_1, T_2 and T_3 (as defined in [Ros11b, Chap. 14]) to 1, 3 and 7, respectively. Similarly, for Fischer's mutual exclusion protocol, we vary the number of process nodes and fix the two time constants to 3 and 20.

We use two families of instances for the alternating bit protocol—ABP-0 and ABP-4—where the parameter reflects the maximum number of messages that any medium can lose in a raw. The algorithm is livelock-free as long as this number is finite, i.e., no infinite sequence of messages gets lost in a row. Hence, ABP-0 defines the protocol on reliable media where no message can get lost, while ABP-4 defines media that can lose up to 4 messages (or acknowledgments) in a row. We also experimented interleaving and piping multiple instances from both families—this corresponds to implementing communication networks based on asynchronous (interleaving) or synchronous (piping) message passing. To illustrate the abbreviations, ABP-0-inter-5 denotes the system obtained by interleaving five instances of ABP with unerring media, and ABP-4-pipe-7 denotes the system obtained by piping seven instances of ABP with media loosing up to 4 messages in a row. Piping means redirecting the output of a process to the input of another one and is defined in terms of renaming, parallel composition and hiding, see [Ros98, Ros11b] for details. For the sliding window protocol we performed tests on three families of instances SWP-1, SWP-2 and SWP-3, where the parameter indicates the size of the window. We use a similar naming convention for denoting communication systems obtained by interleaving or piping several instances of the protocol.

4.10 Conclusion and Future Work

In this chapter, we introduced a novel static analysis framework for establishing livelock freedom of CSP processes. Our framework employed a collection of rules on the syntactic structure of a process to either soundly classify a process as livelock free or report an inconclusive result, thereby compromising accuracy for speed. We proposed a general framework, based on reasoning about fixed points in terms of metric spaces, and a framework for finite-state processes, grounded upon analysis of fairness and co-fairness of visible events. Experiments on a wide suite of benchmarks indicated that our approach, combined with state-of-the-art symbolic techniques, outperformed the explicit checker FDR by multiple orders of magnitude, while maintaining a high level of accuracy. We already incorporated the framework for structurally-finite state processes in FDR.

We can follow several avenues for future work.

An interesting property of our approach is the possibility for our algorithm to produce a *certificate* of livelock freedom, consisting among others in the various sets supporting the final judgement. Such a certificate could then be checked by an independent tool.

Alternatively, in cases where livelock freedom cannot be established, we can provide debug information to the user. For systems that are $\overline{\mathsf{SFS}}$, we can identify, for example, all processes whose δ flag is false, and output, for each of those, the collection of fair/co-fair sets of events computed by our tool. For infinite-state systems, we can in addition output the collection of contractive sets $\mathsf{C}_X(P)$ for any subprocess $\mu X \bullet P$ whose set of fair sets $\mathsf{F}(\mu X \bullet P)$ has been found to be empty. This would enable the user to analyse whether certain patterns of behaviours match the intended behaviour of the system and its subcomponents.

Other directions for future work include improving the efficiency of SLAP by incorporating various abstractions such as collapsing all events on a given channel, or placing *a priori* bounds on the size of sets. Conversely, we plan to try to increase accuracy at modest computational cost, for example by making use of algebraic laws at the syntactic level, such as bounded unfoldings of parallel compositions. The framework for reasoning about structurally finite-state processes can be viewed as an instance of abstract interpretation

[CC77, CC02] and in this case perhaps narrowing operators could also be beneficial for enhancing the precision of our tools.

Another interesting follow-up of our work is to try and generalise that an entire class of systems is livelock-free, where systems in the class are parametric in the number of identical process subcomponents. Examples include $\{\text{Milner-}n \mid n \in \mathbb{N}\}$, $\{\text{SWP-1-pipe-}n \mid n \in \mathbb{N}\}$, etc. This would involve analysing systems that spawn n subcomponents for any nondeterministically chosen n.

As a more ambitious goal, we would like to investigate more closely how we can transfer our analyses to other concurrent formalisms such as other process algebras or maybe even shared-variable frameworks. Moreover, we would like to study more carefully the links between livelock freedom and general termination analysis [CY10].

We have ideas how to formulate systems of rules for conservatively establishing *deadlock* freedom of CSP processes. This is intended as an extension of the framework presented in [Mar96], which is implemented in the tool Deadlock Checker [Mar95, MJ97]. The framework requires as input a parallel network of normalised deadlock-free processes which communicate only pairwise [RB85, RD87]:

$$\mathit{Network} = \;\big|\big|_{i \in I}\, \alpha_i \;@\; P_i.$$

The algorithm is able to soundly classify the entire network as deadlock free based on analysis of the so-called state dependency digraph of the network. In this digraph, nodes are modelled by pairs (i, s), where $i \in I$ is an index of a process in the network and s is a state of the process P_i. There is an edge $(i, s) \longrightarrow (j, q)$ in the digraph if and only if $i \neq j$, $A_{ij} = \alpha_i \cap \alpha_j \neq \emptyset$ and the parallel composition $P_i \underset{A_{ij}}{\|} P_j$ can reach a configuration (s, q) such that P_i in state s is limited to performing events from A_{ij} only and the pair (s, q) refuses the whole of A_{ij}. In other words, an edge $(i, s) \longrightarrow (j, q)$ models a potential ungranted strong request from P_i in s to P_j in q. The state dependency digraph can be constructed by only analysing individual processes and parallel compositions of pairs of processes and is of size at most the *sum* of the sizes of the component processes (as opposed to their product). Since a deadlock scenario necessarily results in a cycle in the state dependency digraph of the network, the absence of cycles in the digraph soundly (although incompletely) guarantees deadlock freedom of the network. We intend to extend this algorithm in two ways. Firstly, we plan to devise a system of rules that enables reasoning about an arbitrary structurally-finite state process, without restrictions on the hierarchical structure of the process and the number of processes that a given event can be shared among. Secondly, a cycle in the state dependency digraph indicates a potential deadlock only if every process contributes at most a single node on this cycle. In general, however, checking whether the digraph contains a cycle of this type is NP-complete. Hence, adopting such an algorithm would improve on accuracy but worsen performance. Nevertheless, similarly to slap, we believe that employing symbolic techniques, in particular SAT solving, would be beneficial for overcoming this bottleneck and would result in significant improvements in terms of accuracy while incurring only minor performance penalties.

Chapter 5

Abstraction Schemes and CEGAR Framework for CSP and FDR

5.1 Introduction

5.1.1 Abstraction

Broadly speaking, abstraction [CGL94, CGP99, BK08] is a mechanism for regulating the amount of detail present in the system. It is a powerful state-space reduction technique, invaluable in practice for tractably verifying industrial-scale applications, even infinite-state ones. In the context of model checking, abstraction techniques have proven extremely beneficial in cases where the specification refers only to certain aspects of the system under consideration, drawing away from low-level details or from the behaviours of entire components. Abstraction can then serve as a vehicle for extracting and narrowing the focus mostly on those details that are relevant to the specific property, disregarding or simplifying the irrelevant information, thereby reducing the size of the model.

In verification, and in real life, it is rarely the case that a single specification targets to cover *all* possible notions of correctness of a system with a single shot—this approach is rather unintuitive, error-prone and, furthermore, computationally expensive. Much in the style of divide-and-conquer, the verification process is usually incremental in the sense that the system is verified against a number of independent correctness properties that individually often do not address the full spectrum of details. Therefore, abstraction techniques are quite versatile and applicable to many different contexts in practice.

In general, we distinguish between exact and inexact abstractions. *Exact* abstractions preserve all kinds of properties of the system under consideration, i.e., a property holds in the concrete system precisely whenever it holds in the abstraction of the system. With *inexact* abstractions, on the other hand, the concrete and the abstract model of the system may satisfy a different set of properties. An abstraction is *conservative* with respect to a certain property φ if it soundly preserves φ, i.e., if it guarantees that whenever φ holds in the abstract model, then φ also necessarily holds in the concrete model of the system. Generally, abstractions that yield overapproximations of the behaviours of the system soundly preserve safety properties—properties of the form "something bad will never happen". In contrast, abstractions resulting in underapproximations are conservative with respect to liveness properties, which, broadly speaking, specify that "something good will eventually happen". The rationale behind using conservative abstractions is that they compromise precision for efficiency—they generate more compact state spaces and are more powerful for tackling the state-space explosion problem.

In practice, abstractions are often induced by certain partitions (equivalence relations) on the concrete state spaces of the model of a system (see Section 2.1.3.2). Given a model and a partition on its state space, existential and universal abstraction [CGL94, Kur94] are uniquely defined and yield, respectively, an overapproximation and an underapproximation of the behaviours of the concrete system.

We focus mainly on existential abstraction [CGL94] which yields an overapproximation of the behaviours of the system and hence guarantees to preserve safety properties. If a safety specification holds in the abstract system, then it also holds in the concrete system. However, if the specification is violated in the abstract system, the counterexample generated might be *spurious*, i.e., it might correspond to a behaviour which is not present in the concrete system. In this case, the abstraction must be refined in a way that the spurious example be eliminated.

In general, coarser existential abstractions yield a more compact state space, but also introduce a greater number of spurious counterexamples. Too refined abstractions, on the other hand, are less likely to introduce spurious counterexamples, but suffer more severely from the state-space explosion problem. Hence, hitting the right balance in granularity is essential.

5.1.2 Counterexample-Guided Abstraction Refinement

Counterexample-guided abstraction refinement (CEGAR)[CGJ$^+$00, CGJ$^+$03, Kur94] is an automated iterative abstraction-refinement technique for establishing the correctness of safety properties. Firstly, a coarse conservative *initial abstraction* of the system is generated. If model checking the abstraction against the specification fails and the counterexample is spurious, the counterexample is used to automatically produce a finer abstraction of the system. Thus, the specification is model-checked against a sequence of increasingly refined abstractions until either the property is proven to hold or a genuine counterexample is derived. The process of checking whether the counterexample is spurious (i.e., is a "false positive") is called *counterexample validation* and the process of producing a finer abstraction for eliminating the spurious counterexample is called *abstraction refinement.* It is important to note that all stages of CEGAR can be completely automated and hence no human support is required. A general issue to be careful about is the guarantee for termination of the CEGAR loop—the sequence of refined abstractions has to converge in a finite number of steps to a system which is equivalent to the original one with respect to some equivalence relation.

5.1.3 Introducing a CEGAR Framework for CSP and FDR

Building upon FDR, we develop a series of abstraction/refinement schemes for the traces, stable-failures and failures-divergences models of CSP and embed them into a fully automated and compositional CEGAR framework.

Since CSP refinement is monotonic and transitive (see Section 2.2.3), we carry out the stages of initial abstraction, counterexample validation and abstraction refinement component-wisely on the level of the transition systems of the sequential leaf processes. This is facilitated by the fact that supercombinators act on a higher level to control how leaf processes interact. The only point at which we consider the global state space is during the verification phase but even then, FDR only explores the compact state space induced by the abstracted leaf processes and, moreover, performs this on-the-fly.

Extending the original CEGAR framework [CGJ$^+$00, CGJ$^+$03], our abstractions are identified by partitions of the state spaces of the leaf processes, coupled with minimal existential abstraction and annotations for preserving nondeterminism-related information.

In general, traditional existential abstraction is inadequate for preserving liveness properties; in particular, in our setting it underapproximates refusal sets instead of overapproximating them. To handle this, we extend the approach used in the tool MAGIC [CCOS04, CCO$^+$05] to define abstract minimal acceptances and abstract divergences. We also devise a number of strategies for refining the resulting spurious abstract behaviours.

FDR plays a crucial role in the compositional handling of counterexample validation and abstraction refinement. First of all, upon detecting an erroneous behaviors of the overall process, FDR provides contribution behaviours of the individual leaf processes that can be validated separately. For establishing whether or not an abstract counterexample behaviour is spurious, model checkers generally employ explicit or symbolic simulation techniques or theorem provers. Our approach is to devise instead a suitable refinement check in FDR and carry out this on leaf-process level. Similarly, we employ refinement checks FDR in order to obtain all abstract executions of a given spurious behaviour. Consequently, we can efficiently validate and eliminate, if necessary, all types of behaviours observable in the three semantic models of interest—finite and lasso traces, divergence traces and stable failures.

Regarding abstraction and abstraction refinement, we follow and extend algorithms for obtaining bisimulation quotients based on the iterative partition refinement framework [CGP99, BK08]. The series of finer abstractions that we construct converge, in the worst case, to a strong (or DRW) bisimulation quotient of the LTS of the implementation process in question (see Section 2.2.4.3). Generally, we adopt lazy refinement strategies that yield coarser abstractions even though it takes a greater number of iterations to converge.

Preliminary experiments with the CEGAR framework indicate a significant enhancement in terms of performance when verifying both safety and liveness properties, including checks for livelock and deadlock. High performance, however, depends on whether or not the specification property is established or refuted before all leaf processes get fully expanded.

5.1.4 Related Work

To the best of our knowledge, the only application of CEGAR to the CSP setting is the one in [Low04]. However, the technique described there is not automatic—initial abstraction and consecutive refinements are devised by hand and brain on the higher CSP level. In contrast, our CEGAR framework is completely automated and initial abstraction and subsequent abstraction refinements are carried out on the operational representation of the sequential components.

A fully compositional CEGAR framework for analysing concurrent C programs communicating via synchronous message passing was introduced in the tool MAGIC [COYC03, CCO+05]. The MAGIC framework handles parallel composition of sequential programs in a very CSP-like process-algebraic manner to verify safety [COYC03] and liveness [CCO+05] properties. Based on compositionality theorems for the *alphabetised parallel operator* proven in [Ros98], MAGIC incorporates a fully automatic CEGAR loop in which all operations of initial abstraction, counterexample validation and abstraction refinement are carried out on the sequential C programs and not on the entire parallel composition. For tackling liveness properties, new notions of *abstract refusals* and *abstract failures* are introduced [CCO+05], which we also adopt and extend. Those are necessary because standard abstraction schemes do not preserve liveness properties, and in particular, they underapproximate refusal sets instead of overapproximating them. Regarding the termination of the CEGAR loop, for both safety and liveness properties, in the worst case the sequence of increasingly refined overapproximations converges in a finite number of steps to a system that is bisimilar to the original one.

The CEGAR schemes converging to bisimulation quotients that we incorporate are extensions of those techniques. Differences and challenges in our settings are due to the presence of silent τ actions and the need to deal with a richer collection of compositional operators including hiding, renaming, choice, etc. Furthermore, regarding parallel composition, we need to be able to handle the *generalised parallel* operator by which sequential components synchronise on a given set of events and not on the intersection of their alphabets as in alphabetised parallel. This makes our approach a bit less conservative in that spurious counterexamples might not get completely eliminated from the system after refining the abstractions—spurious behaviours might remain present owing to alternative contributions of the component processes. Nevertheless, the abstraction-refinement phase always produces a proper refinement of the abstraction, which provides a guarantee for termination. We note, however, that in the general case, the rate of convergence of our CEGAR loop might be considerably lower compared to the approaches in [COYC03] and [CCO+05].

In general, the CEGAR framework has been more actively investigated in the context of shared-variable programs and state-based formalisms. The original proposal [CGJ$^+$00, CGJ$^+$03] was initially integrated in the BDD engine of the model checker NuSMV [CCGR00] and evaluated on a benchmark of hardware circuits. A CEGAR framework based on *predicate abstraction* [GS97] was first implemented as a part of the model checker Slam [BR01, BLR11]. The framework supports the verification of safety properties of C programs and focuses on the sequential model of computation. Extensions of Slam with limited support for concurrency include [QW04], where the execution of multiple threads within a bounded number of context switches is encoded into a sequential program. SatAbs [CKSY05] is a verification platform for reasoning about for C/C++ programs and it also employs a CEGAR framework based on predicate abstraction. SatAbs offers support for reasoning about shared-variable multi-threaded programs by adopting partial-order reductions [CKS05], and recently also by exploiting techniques based on symmetry reductions [DKKW11]. Another prominent model checker that provides support for verifying multi-threaded C programs is Blast [HJMS02, HJMQ03], which embeds into its CEGAR framework lazy predicate abstraction coupled with assume-guarantee reasoning . The model checker boom [BMWK10] verifies concurrent Boolean programs spawning dynamic but bounded number of threads; it achieves this by employing counter abstraction.

5.2 General Skeleton of the CEGAR Framework for CSP and FDR

In this section, we outline the general skeleton for CEGAR that we use in all three semantic CSP models of interest. We also highlight compositionality aspects, as well as discuss convergence rates and guarantees.

Let us suppose that we want to establish that $\mathit{Spec} \sqsubseteq_{\mathcal{M}} P$ for $\mathcal{M} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$. We assume that after supercompilation $P = (SC_P, \langle P_1, \ldots, P_n \rangle)$, where $P_1, \ldots, P_n$ are the sequential leaf processes of P, and SC_P is the set of all supercombinator rules, i.e., all rules mapping actions of a collection of those leaf processes to a resulting action of the high-level process P. Let us recall that the list of leaf processes together with the set of supercombinators is a complete characterisation of the high-level process P (see Section 2.3.1.2). Hence,

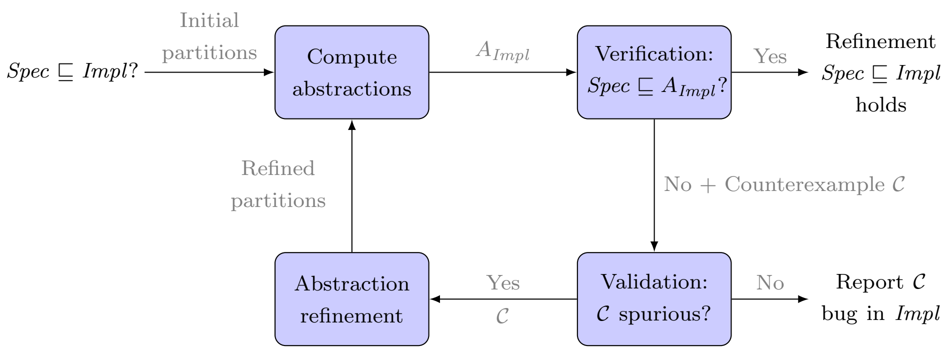

Spec ⊑ Impl?
Initial partitions
Compute abstractions
A_{Impl}
Verification: $Spec \sqsubseteq A_{Impl}$?
Yes
Refinement $Spec \sqsubseteq Impl$ holds
Refined partitions
No + Counterexample $\mathcal{C}$
Abstraction refinement
Yes
$\mathcal{C}$
Validation: $\mathcal{C}$ spurious?
No
Report $\mathcal{C}$ bug in $Impl$

Figure 5.1: CEGAR framework

to simplify the notation below, we will write $P = P_1 \parallel \ldots \parallel P_n$ even though the process tree of P may contain any CSP operator other than recursion.

As CSP refinement is monotonic and transitive (see Section 2.2.3), we carry out the main stages of initial abstraction, counterexample validation and abstraction refinement component-wisely on leaf-process level, facilitated by the capabilities of FDR. This is justified by the fact that supercombinators act on a higher level to control how leaf processes interact and that strong bisimulation (as well as DRW bisimulation, see Section 2.2.4.3) is a congruence for all CSP operators in all semantic models that we consider [Ros11b]. Essentially, we build upon the following two facts:

1. If for all $i \in \{1, \ldots, n\}$, $C_i \sqsubseteq_{\mathcal{M}} P_i \sqsubseteq_{\mathcal{M}} B_i$, then the following refinement relations hold:

$$(SC_P, \langle C_1, \ldots, C_n \rangle) \sqsubseteq_{\mathcal{M}} P = (SC_P, \langle P_1, \ldots, P_n \rangle) \sqsubseteq_{\mathcal{M}} (SC_P, \langle B_1, \ldots, B_n \rangle). \quad (5.1)$$

2. For any CSP operator $\otimes$ other then recursion, any CSP processes P, Q, P' and Q', and any strong (or DRW) bisimulation relation $\sim$,

$$\text{if } P \sim P' \text{ and } Q \sim Q', \text{ then } P \otimes Q \sim P' \otimes Q'. \quad (5.2)$$

The only point at which we consider the global state space of the system is during the refinement checking phase, and even then, we only need to construct (on-the-fly) the smaller state space induced by the abstracted leaf processes.

Regarding abstraction and abstraction refinement, the algorithm for all models is based on the iterative partition refinement framework [CGP99, BK08].

In the remainder of this section, we briefly outline the major stages of our CEGAR loop, which is depicted in Figure 5.1. We use the definitions of generalised labelled transition systems (GLTS's), minimal acceptances and divergence labellings as defined in [Ros94, RGG$^+$95] and described in Section 2.3.2. Let us recall that in a GLTS representation of a process every process state is labelled with a set (an antichain) of minimal acceptance sets and a flag indicating whether the state is divergent.

Preprocessing Step. In order to simplify the algorithms, prior to embarking on the CEGAR loop, we transform the transition system of each leaf P_i into a semantically equivalent τ-free (generalised) transition system. We do that by applying diamond elimination compression [RGG$^+$95, G$^+$05] with the help of FDR (see Section 2.3.3.1). The transition system of each P_i after diamond compression has the following properties:

- it has at most as many nodes as the original transition system,
- it is free of τ-transitions,
- as opposed to post normalisation, there still might be nondeterminism left as far as visible actions are concerned,
- all τ-related nondeterminism of a node is stored as an annotation of the node, which consists of a divergence flag and a set of minimal-acceptance sets (see Section 2.3.2 and Section 2.3.3).

This preprocessing transformation is relatively cheap in time and space: for each leaf P_i, diamond elimination can be performed in time linear in the size of P_i to obtain a semantically equivalent process with at most as many states as P_i. We note, however, that diamond elimination typically results in an increase of the size of the minimal-acceptance annotations.

Initial Abstraction. To generate an initial abstraction A^0 of P, for each leaf P_i, we aggregate together states of P_i according to some criteria, e.g., depending on the semantic model or the degree of granularity we want to start with (we discuss different strategies in Section 5.6.1 but, in general, any partition would work). The resulting partition Π_i^0, combined with (minimal) existential abstraction and adjustment of the divergence flag and the set of minimal acceptances, uniquely identifies the abstract process A_i^0 and guarantees that $A_i^0 \sqsubseteq_{\mathcal{M}} P_i$. We construct the initial abstraction A^0 of P then as follows:

$$A^0 = (SC_P, \langle A_1^0, \dots, A_n^0 \rangle).$$

We note that for generating A^0 we use the set of supercombinators SC_P of P, but for simplicity, we will write $A^0 = A_1^0 \parallel \dots \parallel A_n^0$. Then, due to compositionality theorems proven in [Ros98] and formalised as (5.1),

$$\underbrace{A_1^0 \parallel \dots \parallel A_n^0}_{A^0} \;\sqsubseteq_{\mathcal{M}}\; \underbrace{P_1 \parallel \dots \parallel P_n}_{P},$$

and hence performing the initial abstraction component-wisely is sound.

Refinement Checking. Let us suppose that at some iteration k of the CEGAR loop we obtain an abstract process $A^k = (SC_P, \langle A_1^k, \dots, A_n^k \rangle) = A_1^k \parallel \dots \parallel A_n^k$ that satisfies the invariant:

$$\forall\, i \in \{1, \dots, n\}, A_i^0 \sqsubseteq_{\mathcal{M}} A_i^k \sqsubseteq_{\mathcal{M}} P_i,$$

and therefore also:

$$\underbrace{A_1^0 \parallel \dots \parallel A_n^0}_{A^0} \;\sqsubseteq_{\mathcal{M}}\; \underbrace{A_1^k \parallel \dots \parallel A_n^k}_{A^k} \;\sqsubseteq_{\mathcal{M}}\; \underbrace{P_1 \parallel \dots \parallel P_n}_{P}.$$

If *Spec* $\sqsubseteq_{\mathcal{M}} A^k$, by transitivity of $\sqsubseteq_{\mathcal{M}}$, *Spec* $\sqsubseteq_{\mathcal{M}} P$ and we are done. In case *Spec* $\not\sqsubseteq_{\mathcal{M}} A_1^k \parallel \dots \parallel A_n^k$, depending on the semantic model $\mathcal{M}$, FDR generates a counterexample behaviour b and a contribution b_i of each leaf process A_i^k for obtaining b. It is important to note that for a given behaviour b, the list of contributions $\langle b_i \mid i \in \{1, \dots, n\} \rangle$ is *not* uniquely identified, and therefore FDR plays a crucial role at this stage.

Counterexample Validation. For checking whether the behaviour b generated by FDR is a valid behaviour of $P = P_1 \parallel \dots \parallel P_n$, we simulate each b_i on P_i, or more precisely, we devise a special refinement check in FDR to infer that. If, for every i, b_i is a valid behaviour of P_i, then b is a genuine counterexample for the refinement *Spec* $\sqsubseteq_{\mathcal{M}} P$—we report the bug and exit the CEGAR loop.

Otherwise, there exists a leaf process P_j for which the behaviour b_j is spurious. We note that, in this case, the behaviour b can still be a valid behaviour of P under some other contributions of the leaves. Hence, our strategy for counterexample validation is conservative. If we declare that the behaviour b is valid, then it is guaranteed to be valid. However, the reverse does not necessarily hold and the subsequent abstraction refinement step may result in the same b being found again, this time decomposed into different b_i's that may or may not witness the validity of b.

Abstraction Refinement. Having established that b_j is a valid behaviour of A_j^k but not of P_j, we refine A_j^k to A_j^{k+1} in a way that eliminates b_j from A_j^{k+1} and maintains the invariant $A_j^k \sqsubseteq_{\mathcal{M}} A_j^{k+1} \sqsubseteq_{\mathcal{M}} P_j$. This is performed by splitting blocks on all paths in A_j^k that accept b_j. The blocks in A_j^k that correspond to the failure states in P_j are split according to their reachable successor blocks, depending on the semantic model and the type of counterexample behaviour. The resulting partition Π_j^{k+1} is a proper refinement of Π_j^k which, after readjusting the existential abstraction, the divergence flag and the set of minimal acceptances, guarantees $A_j^k \sqsubseteq_{\mathcal{M}} A_j^{k+1}$. The partition-refinement step is similar to a unit step in the Paige-Tarjan algorithm [PT87, BK08] and, therefore, subsequent abstraction refinements of A_j^k converge in a finite number of steps to a (strong) bisimulation quotient of P_j [COYC03, CCO+05]. Hence, $A_j^{k+1} \sqsubseteq_{\mathcal{M}} P_j$. Then, due to compositionality theorems,

$$\underbrace{A_1^k \parallel \ldots \parallel A_j^k \parallel \ldots \parallel A_n^k}_{A^k} \sqsubseteq_{\mathcal{M}} \underbrace{A_1^k \parallel \ldots \parallel A_j^{k+1} \parallel \ldots \parallel A_n^k}_{A^{k+1}} \sqsubseteq_{\mathcal{M}} \underbrace{P_1 \parallel \ldots \parallel P_j \parallel \ldots \parallel P_n}_{P},$$

and we can continue with the next iteration $(k+1)$ of the CEGAR loop.

Let us note that the choice of how many leaf processes to refine at each iteration of the CEGAR loop depends on whether we want to adopt a lazy or an aggressive strategy for convergence. Generally, we lazily refine just a single leaf that contributes a spurious behaviour. Nevertheless, we also support an aggressive strategy that refines all leaf processes giving rise to spurious contributions.

Termination. At each iteration of the CEGAR loop, we properly refine a partition of at least one leaf process. Hence, the procedure converges in at most $\mid P_1 \mid + \ldots + \mid P_n \mid$ number of steps, where $\mid P_i \mid$ is the number of states in P_i. As each leaf P_i converges in the worst case to a strong (or DRW) bisimulation quotient of P and the corresponding bisimulation equivalence is a congruence with respect to all CSP operators (5.2), the bisimulation-reduced leaves together with the set of supercombinators SC_P yield a bisimulation quotient of P (perhaps not the coarsest one).

5.3 Abstraction Schemes for CSP

In this section we define our notion of abstraction for all semantic models $\mathcal{T}, \mathcal{F}$ and $\mathcal{N}$. We first list the general requirements that we need any abstraction to conform to. Then we show how to generate different instances of abstraction meeting those requirements depending on the semantic model we want to work in.

We carry out the stages of abstraction and abstraction refinement on the level of transition systems. Hence, in this section we identify processes with their associated labelled transition systems (and possibly their generalised versions). As outlined in Section 2.1.3.2, partitions and equivalence relations are in one-to-one correspondence, as are blocks in partitions and equivalence classes induced by equivalence relations. Hence throughout this section we will use those notions interchangeably.

5.3.1 Requirements

Let us fix a process $P = \langle S, \mathsf{init}, \Sigma, T \rangle$ and a semantic model $\mathcal{M} \in \{\mathcal{T}, \mathcal{F}, \mathcal{N}\}$. We will say that a process $A = \langle S_A, \mathsf{init}_A, \Sigma_A, T_A \rangle$ is an *abstraction* of P in the model $\mathcal{M}$ if A satisfies the following requirements:

1. $\Sigma_A = \Sigma$,

2. A contains at most as many states as P,

3. $A \sqsubseteq_{\mathcal{M}} P$, and more specifically, for every execution $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n$ of P there exists an execution $q_0 \xrightarrow{a_0}_A q_1 \xrightarrow{a_1}_A \ldots q_{n-1} \xrightarrow{a_{n-1}}_A q_n$ of A such that:

    - if $\mathcal{M} \in \{\mathcal{F}, \mathcal{N}\}$, then for every $X \in \mathsf{min\_acceptances}(s_n)$ there exists $Y \in \mathsf{min\_acceptances}(q_n)$ such that $Y \subseteq X$,

    - if $\mathcal{M} = \mathcal{N}$ and s_n is divergent, then q_n is divergent as well.

Typically, A will contain a lot less states and allow a lot more behaviours compared to P. We will refer to states in S and S_A as *concrete states* and *abstract states*, respectively. Similarly, we will call T a *concrete transition relation* and T_A an *abstract transition relation.*

We remark that the absence of τ actions in P simplifies significantly the requirements we set out for A—it lets us reason about executions rather than (semantic) traces.

5.3.2 An Abstraction for the Traces Model

Let us first focus on the traces model exclusively. Similarly to [CGJ+03, COYC03], our abstraction A of P is uniquely identified by a partition of the concrete state space S (which defines the abstract state space S_A) and existential abstraction [CGL94, CGP99] (which defines the abstract transition relation T_A).

Throughout this section we take an arbitrary partition $\sim$ of S. We also view $\sim$ as an equivalence relation $\sim$ on S. Then we generate A as the quotient of P with respect to the equivalence relation $\sim$, coupled with minimal existential abstraction. Formally, $A = \langle S_A, \mathsf{init}_A, \Sigma_A, T_A \rangle$, where:

- $S_A = S/{\sim}$, i.e., S_A is the set $\{[s]^{\sim} \mid s \in S\}$ of all equivalence classes induced in S by $\sim$
- $\mathsf{init}_A = [\mathsf{init}]^{\sim}$
- $\Sigma_A = \Sigma$
- $T_A = \{([s_i]^{\sim}, a, [s_j]^{\sim}) \mid (s_i, a, s_j) \in T\}$. In other words, for all $s_i, s_j \in S$ and $a \in A$, $[s_i]^{\sim} \xrightarrow{a}_A [s_j]^{\sim}$ if and only if there exist $s'_i \in [s_i]^{\sim}$ and $s'_j \in [s_j]^{\sim}$ such that $s'_i \xrightarrow{a} s'_j$.

The transition system we constructed is an instance of *existential abstraction* and, in fact, of *minimal* existential abstraction [CGL94]. Non-minimal existential abstraction can be derived from this construction by substituting T_A by any proper superset of T_A. We will write $P/{\sim}$ to denote the abstraction A of P obtained by the above construction and we will refer to it as the *minimal existential quotient* of P under the equivalence relation $\sim$. We state a result which is very well known in literature.

Proposition 5.3.1 ([CGP99]). *Let $P = \langle S, \mathsf{init}, \Sigma, \longrightarrow \rangle$ be a labelled transition system, $\sim$ be an equivalence relation on S, and $P/\sim$ be the minimal existential quotient of P with respect to $\sim$. If $\mathsf{init} \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots s_{n-1} \xrightarrow{a_{n-1}} s_n$ is an execution of P, then $[\mathsf{init}]^{\sim} \xrightarrow{a_0} [s_1]^{\sim} \xrightarrow{a_1} \dots [s_{n-1}]^{\sim} \xrightarrow{a_{n-1}} [s_n]^{\sim}$ is an execution of $P/\sim$.*

Proof. The proof proceeds by induction on the length of the execution and is a direct consequence of the construction of the existential quotient. □

The following proposition demonstrates that for any equivalence relation $\sim$ on S, $A = P/\sim$ satisfies the general requirements for abstraction that we imposed in Section 5.3.1.

Proposition 5.3.2. *For any labelled transition system P and any equivalence relation $\sim$ on the state space of P, $P/\sim \sqsubseteq_T P$.*

Proof. By Proposition 5.3.1 we can conclude that any trace of P is matched by a trace of $P/\sim$, whereby $\mathsf{traces}(P) \subseteq \mathsf{traces}(P/\sim)$ and hence, $P/\sim \sqsubseteq_T P$. □

Let $\sim$ and $\sim'$ be equivalence relations on the concrete state space S. Let us recall (see Section 2.1.3.2) that $\sim'$ is a refinement of $\sim$ if for all $s \in S$, $[s]^{\sim'} \subseteq [s]^{\sim}$. The refinement relation on equivalence relations on S is then a partial order. We will write $\sim \leq \sim'$ if $\sim'$ is a refinement of $\sim$, and $\sim < \sim'$ if $\sim'$ is a proper refinement of $\sim$. The following proposition states that a refinement of the equivalence relation induces trace refinement of the corresponding minimal existential quotient.

Proposition 5.3.3. *Let P be a labelled transition system and $P/\sim$ be an abstraction of P. For any refinement $\sim'$ of the equivalence relation $\sim$,*

$$P/\sim \; \sqsubseteq_T P/\sim' \; \sqsubseteq_T P.$$

Proof. By Proposition 5.3.2, $P/\sim \sqsubseteq_T P$ and $P/\sim' \sqsubseteq_T P$. We now prove that the refinement $P/\sim \sqsubseteq_T P/\sim'$ also holds, having that for every $s \in S$, $[s]^{\sim'} \subseteq [s]^{\sim}$. In order to establish that, let us define a relation $\equiv$ on the state space $S/\sim'$ of $P/\sim'$. For any $s, q \in S$, we let:

$$[s]^{\sim'} \equiv [q]^{\sim'} \text{ if and only if } s \sim q.$$

In other words, we aggregate back together all those classes of equivalence under $\sim'$ that are obtained by splitting the same equivalence class under $\sim$. Then, since $\sim$ is an equivalence relation on S, $\equiv$ is a well-defined equivalence relation on $S/{\sim'}$. Therefore, by Proposition 5.3.2,

$$(P/{\sim'})/{\equiv} \sqsubseteq_T P/{\sim'} \,. \tag{5.3}$$

Let us further notice that the relation $R = \{([s]^{\sim}, [s]^{\sim'}/{\equiv}\ ) \mid s \in S\}$ is a strong bisimulation relation for $(P/{\sim}, (P/{\sim'})/{\equiv})$ and even more, it defines an isomorphism between the two transition systems. Therefore, $P/{\sim} \equiv_T (P/{\sim'})/{\equiv}$. By this trace equivalence and (5.3) we conclude that $P/{\sim} \sqsubseteq_T P/{\sim'}$. $\square$

The latter proposition justifies the termination of our CEGAR cycle: starting with any equivalence relation (partition) $\sim$ on S, subsequent proper refinements $\sim\, < \,\sim' < \,\sim'' < \,\ldots$ of $\sim$ generate a sequence of increasingly refined processes

$$P/{\sim} \sqsubseteq_T P/{\sim'} \sqsubseteq_T P/{\sim''} \sqsubseteq_T \ldots \sqsubseteq_T P,$$

that converges to P in at most $|\,S\,|$ iterations.

In our implementation for the traces model, we start with the coarsest possible initial abstraction, the one induced by aggregating all states in S in a single equivalence class. We refer to the initial abstraction as A^0.

5.3.3 A Collection of Abstractions for the Stable-Failures Model

Let us fix a process $P = \langle S, \mathsf{init}, \Sigma, T\rangle$ and an equivalence relation (partition) $\sim$ on S. In this section we will introduce a collection of abstractions for the stable-failures model $\mathcal{F}$ that satisfy the general requirements listed in Section 5.3.1.

Unfortunately, the abstraction $P/{\sim}$ that we proposed in the previous section proves inadequate for preserving liveness properties and, hence, for using in the stable-failures model. As first observed in MAGIC [CCOS04, CCO+05], existential abstraction naturally underapproximates refusal sets at abstract states instead of overapproximating them. In the setting of MAGIC, a refusal $\mathsf{Ref}(s)$ of a concrete state $s \in S$ is a set of events and is defined as $\Sigma \setminus \mathsf{initials}(s)$. Following the construction of the existential quotient $P/{\sim}$, $\mathsf{initials}([s]^{\sim}) = \bigcup_{s' \in [s]^{\sim}} \mathsf{initials}(s')$ which suggests that $\mathsf{Ref}([s]^{\sim}) = \bigcap_{s' \in [s]^{\sim}} \mathsf{Ref}(s')$, which is indeed the opposite of what we require for a conservative abstraction. The particular solution proposed in [CCOS04, CCO+05] revolves around the notion of an *abstract refusal* of an abstract state, which is defined as $\mathsf{AbsRef}([s]^{\sim}) = \bigcup_{s' \in [s]^{\sim}} \mathsf{Ref}(s')$.

In FDR, compression functions such as normalisation and diamond elimination merge together sets of states of a process. Hence, after applying a compression, FDR transforms the transition system of a process into a generalised transition system. In the latter, every state is annotated by a flag indicating whether the process is divergent and, if it is not, by an extra *set* of maximal refusals (or minimal acceptances). The maximal refusals (or alternatively, the minimal acceptances) of a generalised state is an antichain over the powerset of Σ under the set-containment order, i.e., is a set of pair-wise incomparable sets of events. The antichain stores maximal-refusal (or alternatively, minimal-acceptance) information about *all* states constituting the generalised state. Therefore, we need to extend the notion of an abstract refusal proposed in MAGIC and adapt it to the setting of a *set* of maximal refusals. We follow the approach chosen by FDR to instead collect the set of minimal acceptances; hence we define a set of abstract minimal acceptances. To recall, a minimal acceptance is the dual of a maximal refusal (see Section 2.2.4.2).

We now devise an abstraction A for the stable-failures model $\mathcal{F}$ that meets the requirement $A \sqsubseteq_F P$. We obtain A from $P/\sim$ by redefining the set of minimal acceptances of an abstract state $[s]^{\sim}$. We need to ensure that:

$$\forall X \in \mathsf{min\_acceptances}(s) \bullet \exists Y \in \mathsf{min\_acceptances}([s]^{\sim}) \bullet Y \subseteq X. \tag{5.4}$$

In the following definitions, let us suppose that $\mathsf{min\_basis}$ is an operator which, given a set of sets of events as an argument, returns the antichain of its minimal sets of events.

Proposal 1. Our first proposal is the following.

$$\mathsf{min\_acceptances}([s]^{\sim}) = \mathsf{min\_basis}(\bigcup_{s' \in [s]^{\sim}} \mathsf{min\_acceptances}(s')) \tag{5.5}$$

In this case, the property (5.4) follows trivially. Indeed, suppose $X \in \mathsf{min\_acceptances}(s)$. Then $X \in \bigcup_{s' \in [s]^{\sim}} \mathsf{min\_acceptances}(s')$. Therefore, there exists $Y \subseteq X$ such that $Y \in \mathsf{min\_basis}(\bigcup_{s' \in [s]^{\sim}} \mathsf{min\_acceptances}(s')) = \mathsf{min\_acceptances}([s]^{\sim})$.

This proposal has the advantage of offering to devise a refinement of the abstraction in a straightforward way, as we will demonstrate in Section 5.5. However in practice employing it turned out to blow-up exponentially the set of minimal acceptances of the high-level process. In fact, our experiments indicated that abstracting traces and abstracting stable failures (using this construction) were triggered by two opposite forces: the coarser the equivalence relation, the better the abstraction in the traces model and the worse in the stable-failures model (and conversely).

Proposal 2. Our second proposal is intended to fix this flaw by trying to decrease the number and size of the abstract minimal acceptances that we generate. We do so by fully expanding minimal acceptances of singleton abstract states and being as conservative as possible in all other cases. Notice that if $[s]^{\sim}$ is not a singleton, then $\mathsf{min\_acceptances}([s]^{\sim})$ is a singleton, i.e., it contains just a single minimal-acceptance set.

$$\mathsf{min\_acceptances}([s]^{\sim}) = \begin{cases} \mathsf{min\_acceptances}(s) & \text{if } [s]^{\sim} \text{ is a singleton} \\ \{\bigcap_{s' \in [s]^{\sim}} \bigcap_{A_k \in \mathsf{min\_acceptances}(s')} A_k\} & \text{otherwise} \end{cases} \tag{5.6}$$

This proposal, however, might be overly conservative in certain occasions. If an abstract state $[s]^{\sim}$ is composed of multiple concrete states with totally incompatible minimal acceptances, $\mathsf{min\_acceptances}([s]^{\sim})$ can easily become the singleton set containing the empty set, which does not contain any useful information.

Proposal 3. Our third proposal is intended to maintain a balance between the two proposals suggested above:

$$\mathsf{min\_acceptances}([s]^{\sim}) = \begin{cases} \mathsf{min\_acceptances}(s) & \text{if } [s]^{\sim} \text{ is a singleton} \\ \mathsf{min\_acceptances}(s_k) & \text{if } |\,[s]^{\sim}\,| > 1 \text{ and } \exists\, s_k \in [s]^{\sim} \bullet \\ & \forall\, s' \in [s]^{\sim} \bullet s_k \sqsubseteq_f s' \\ \{\bigcap_{s' \in [s]^{\sim}} \bigcap_{A_k \in \mathsf{min\_acceptances}(s')} A_k\} & \text{otherwise} \end{cases} \tag{5.7}$$

The notation $s_k \sqsubseteq_f s'$ is derived from $s_k \sqsubseteq_F s'$ by ignoring the initial events available in both states: $s_k \sqsubseteq_f s'$ if for all $X \in \mathsf{min\_acceptances}(s')$ there exists $Y \in \mathsf{min\_acceptances}(s_k)$ such that $Y \subseteq X$.

5.3.4 Extensions for the Failures-Divergences Model

In order to generate an abstraction for the failures-divergences model, we need to further overapproximate the divergence flag of each abstract state. We stipulate that an abstract state is divergent if any of its concrete representatives is divergent:

$$[s]^{\sim} \Uparrow \text{ iff } \exists\, s' \in [s]^{\sim} \centerdot s' \Uparrow .$$

This guarantees that $s \Uparrow$ implies $[s]^{\sim} \Uparrow$, which is a prerequisite for establishing $A \sqsubseteq_{FD} P$. Regarding the minimal acceptances of abstract states,

$$\mathsf{min\_acceptances}([s]^{\sim}) = \begin{cases} \emptyset & \text{if } [s]^{\sim} \Uparrow \\ (5.5) \text{ or } (5.6) \text{ or } (5.7) & \text{otherwise} \end{cases} \tag{5.8}$$

The initial abstraction in this case can be induced by a partition of the state space that arranges all divergent states in one block and all non-divergent states in another (provided both blocks are non-empty).

5.4 Refinement Checking and Counterexample Validation

Let us suppose that at some iteration of the CEGAR loop we obtain the abstracted process $A = A_1 \parallel \ldots \parallel A_n \sqsubseteq_{\mathcal{M}} P = P_1 \parallel \ldots \parallel P_n$, where for every $1 \leq i \leq n$, $A_i \sqsubseteq_{\mathcal{M}} P_i$. If $Spec \sqsubseteq_{\mathcal{M}} A_1 \parallel \ldots \parallel A_n$, by transitivity of $\sqsubseteq_{\mathcal{M}}$, $Spec \sqsubseteq_{\mathcal{M}} P = P_1 \parallel \ldots \parallel P_n$—we exit the CEGAR loop and declare that the refinement holds.

In case $Spec \not\sqsubseteq_{\mathcal{M}} A_1 \parallel \ldots \parallel A_n$, depending on the semantic model $\mathcal{M}$, FDR generates a counterexample behaviour b and a contribution b_i of each leaf process A_i for obtaining b. It is important to note that for a given behaviour b, the list of contributions $\langle b_i \mid 1 \leq i \leq n \rangle$ is *not* uniquely identified, and therefore FDR plays a crucial role at this stage.

In order to establish that b is a valid behaviour of $P = P_1 \parallel \ldots \parallel P_n$, we need to verify that each b_i of A_i is a valid behaviour of its corresponding leaf P_i. If so, we report b as a bug and exit the CEGAR loop. Otherwise, there exists a leaf process P_j for which the behaviour b_j is spurious.

We remark that the reverse implication does not necessarily follow: if there exists a leaf process P_j for which the behaviour b_j is spurious, the overall behaviour b can still be a valid behaviour of P under some other contributions of the leaves. Hence, our strategy for counterexample validation is conservative.

In the remainder of this section, we focus on strategies for determining whether a behaviour b of an abstraction $A = \langle S_A, \mathsf{init}_A, \Sigma, T_A \rangle$ is valid or spurious for a leaf process $P = \langle S, \mathsf{init}, \Sigma, T \rangle$. We skip the leaf indices of P, A and b to keep the notation simpler.

5.4.1 Types of Counterexample Behaviours

In the most expressive of all three models, the failures-divergences model, there can be four types of contributions to erroneous behaviours:

1. A bad initial event giving rise to an illegal trace.
2. A bad minimal-acceptance set resulting in an erroneous stable failure.
3. A bad divergence flag giving rise to a prohibited divergence trace.
4. A bad lasso trace of a subprocess that makes the overall process diverge. We remark that a lasso trace is a finite representation of an infinite trace $\alpha = s^\frown t^\omega$ in which a certain non-empty sequence of visible events t is repeated uninterruptedly and infinitely often. A subprocess P_i of P may contribute the lasso trace α to a divergence of P if the whole set of events encountered in t is hidden in P (and no other process prevents P_i from performing t^ω, or requires to interleave in between events not in t).

We recall that due to the BFS mode of state-space traversal employed in FDR for carrying out the refinement check, every counterexample behaviour is somehow minimal. We will assume then that all four types of counterexample behaviours are based on a valid execution trace:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \quad \cdots \quad s_{n-1} \xrightarrow{a_{n-1}} s_n.$$

By valid we mean that for every $0 \leq i \leq s_{n-1}$, the initials, the divergence flag and the minimal acceptances at every state s_i do not make the overall system exhibit an illegal behaviour. We also note that in case of a counterexample, FDR provides the execution path $\pi = \langle s_0, \dots, s_n \rangle$ as well. On top of this valid behavioural base, we describe the specifics of each erroneous behaviour according to its type:

bad initial An erroneous finite trace $\langle a_0, \dots, a_{n-1}, a_n \rangle$ caused by a transition $s_n \xrightarrow{a_n} s_{n+1}$.

bad minimal acceptance A (too small) set of events $B \in \mathsf{min\_acceptances}(s_n)$ after following the trace $\langle a_0, \dots, a_{n-1} \rangle$.

bad divergence An illegal divergence trace $\langle a_0, \dots, a_{n-1} \rangle$, i.e., the divergence flag δ at state s_n is set to true.

bad lasso trace A prohibited infinite trace $\langle a_0, \ldots, a_{r-1}\rangle^\frown\langle a_r, \ldots, a_n\rangle^\omega$ resulting from a back transition $s_n \xrightarrow{a_n} s_r$ for some $0 \leq r \leq n$.

Let us note that in the traces model, a counterexample behaviour can only be a bad initial, whereas in the stable-failures—a bad initial or a bad minimal acceptance.

5.4.2 Counterexample Validation of Finite Traces, Minimal Acceptances and Divergences

If the counterexample behaviour b is of type bad initial, bad minimal acceptance or bad divergence, we carry out the validation process in the following manner. First we construct a deterministic labelled transition system (alternatively, a normalised process) $N(b, \mathcal{M})$ that allows all possible behaviours in the corresponding model $\mathcal{M}$ but b. Then we use FDR to run a refinement check $N(b, \mathcal{M}) \sqsubseteq_{\mathcal{M}} P$. If the refinement relation holds, then the original leaf process P does not exhibit the behaviour b in the model $\mathcal{M}$, and therefore b is spurious. Alternatively, if the refinement relation is refuted, then we can conclude that b is a genuine behaviour of P. Let us observe that in the latter case, the counterexample behaviour returned by FDR is necessarily precisely b. We exploit a similar refinement check during the abstraction-refinement phase later on to get hold of and eliminate completely all instances of a spurious behaviour b from an abstraction A.

It is important to note that in all three cases of behaviours, $N(b, \mathcal{M})$ is normalised by construction. Therefore, we do not need to normalise it prior to carrying out the refinement check $N(b, \mathcal{M}) \sqsubseteq_{\mathcal{M}} P$. Moreover, because $N(b, \mathcal{M})$ is intended to capture all possible behaviours but one, it is amenable to a design in a compact state space. Hence the validation process proves to be quite efficient in practice.

We now describe the constructions of $N(b, \mathcal{M})$ for the different types of behaviours, assuming that $\mathcal{M} = \mathcal{N}$, i.e., we present the most general constructions. In all the figures illustrating the constructions, states are annotated with their sets of minimal acceptances and divergent states are marked in circles. To keep the notation in the figures simple, we write $\Sigma - a_i$ to denote $\Sigma - \{a_i\}$.

Case Bad Initial. Let first of all consider the case when b is a bad initial, i.e., b is of the form

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \quad \cdots \quad s_{n-1} \xrightarrow{a_{n-1}} s_n \xrightarrow{a_n} s_{n+1},$$

and the "bad event" is a_n. Then $N(b, \mathcal{N}) = \langle S_N, \mathsf{init}_N, \Sigma_N, T_N \rangle$, where (see Figure 5.2):

- $S_N = \{q_0, \ldots, q_{n+1}\}$ and $\mathsf{init}_N = q_0$ and $\Sigma_N = \Sigma$,
- $T_N = \{q_i \xrightarrow{a_i} q_{i+1} \mid 0 \leq i \leq n-1\} \cup \{q_i \xrightarrow{b} q_{n+1} \mid 0 \leq i \leq n \text{ and } b \in \Sigma - \{a_i\}\} \cup \{q_{n+1} \xrightarrow{b} q_{n+1} \mid b \in \Sigma\}$,
- $\mathsf{min\_acceptances}(q_i) = \{\emptyset\}$,
- $\delta(q_i) = \begin{cases} \text{true} & \text{if } i = n+1 \\ \text{false} & \text{otherwise} \end{cases}$

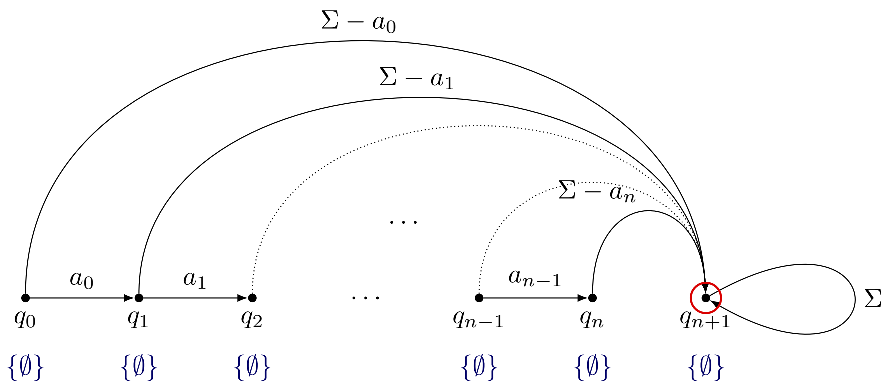


Figure 5.2: $N(b, \mathcal{N})$ for b bad initial a_n after $\langle a_0, \ldots, a_{n-1} \rangle$

Case Bad Minimal Acceptance. Let us assume that b allows the bad minimal acceptance $B \subseteq \Sigma$ after the trace $\langle a_0, \ldots, a_{n-1} \rangle$. Then we construct $N(b, \mathcal{N}) = \langle S_N, \mathsf{init}_N, \Sigma_N, T_N \rangle$, where (see Figure 5.3):

- $S_N = \{q_0, \ldots, q_{n+1}\}$ and $\mathsf{init}_N = q_0$ and $\Sigma_N = \Sigma$,
- $T_N = \{q_i \xrightarrow{a_i} q_{i+1} \mid 0 \leq i \leq n-1\} \cup \{q_i \xrightarrow{b} q_{n+1} \mid 0 \leq i \leq n-1 \text{ and } b \in \Sigma - \{a_i\}\} \cup \{q_n \xrightarrow{b} q_{n+1} \mid b \in \Sigma\} \cup \{q_{n+1} \xrightarrow{b} q_{n+1} \mid b \in \Sigma\}$,

- $\mathsf{min\_acceptances}(q_i) = \begin{cases} \{\{c\} \mid c \in \Sigma - B\} & \text{if } i = n \\ \{\emptyset\} & \text{otherwise,} \end{cases}$
- $\delta(q_i) = \begin{cases} \text{true} & \text{if } i = n+1 \\ \text{false} & \text{otherwise} \end{cases}$

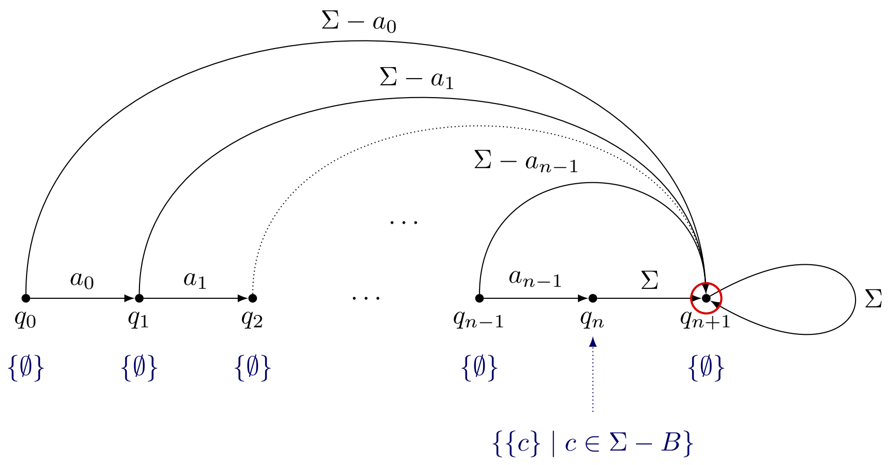


Figure 5.3: $N(b, \mathcal{N})$ for b bad minimal acceptance B after $\langle a_0, \dots, a_{n-1}\rangle$

Case Bad Divergence. Let us assume that b is the divergence trace $\langle a_0, \dots, a_{n-1}\rangle$. Then we construct $N(b, \mathcal{N}) = \langle S_N, \mathsf{init}_N, \Sigma_N, T_N\rangle$, where (see Figure 5.4):

- $S_N = \{q_0, \dots, q_{n+1}\}$ and $\mathsf{init}_N = q_0$ and $\Sigma_N = \Sigma$,
- $T_N = \{q_i \xrightarrow{a_i} q_{i+1} \mid 0 \leq i \leq n-1\} \cup \{q_i \xrightarrow{b} q_{n+1} \mid 0 \leq i \leq n-1 \text{ and } b \in \Sigma - \{a_i\}\} \cup \{q_{n+1} \xrightarrow{b} q_{n+1} \mid b \in \Sigma\}$,
- $\mathsf{min\_acceptances}(q_i) = \{\emptyset\}$,
- $\delta(q_i) = \begin{cases} \text{true} & \text{if } i = n+1 \\ \text{false} & \text{otherwise} \end{cases}$

5.4.3 Counterexample Validation of Lasso Traces

The validation of a lasso trace is of different nature compared to the validation approaches described in the previous section. In this case the counterexample behaviour b is of the

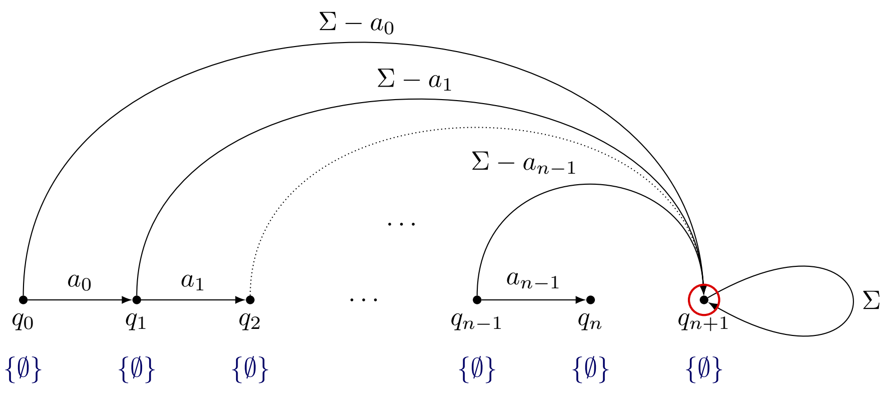


Figure 5.4: $N(b, \mathcal{N})$ for b bad divergence after $\langle a_0, \ldots, a_{n-1} \rangle$

form $\langle a_0, \ldots, a_{r-1} \rangle ^\frown \langle a_r, \ldots, a_n \rangle^\omega$ and is depicted in Figure 5.5. We employ the following algorithm:

1. We construct a process $B(b, \mathcal{N})$ as the most nondeterministic livelock-free process whose traces are precisely all prefixes of $\langle a_0, \ldots, a_{r-1} \rangle ^\frown \langle a_r, \ldots, a_n \rangle^\omega$.

2. We devise a process $P \underset{\Sigma}{\parallel} B(b, \mathcal{N})$. Its traces are precisely all prefixes of $\langle a_0, \ldots, a_{r-1} \rangle ^\frown \langle a_r, \ldots, a_n \rangle^\omega$ that are also traces of P.

3. We check whether the process $\mathit{Test} = (P \underset{\Sigma}{\parallel} B(b, \mathcal{N})) \setminus \Sigma$ is livelock-free. If it is, we conclude that P does not exhibit b, hence b is spurious. If Test is divergent, then there are two possibilities: either b is a valid behaviour of P, or P itself diverges on a prefix of $\langle a_0, \ldots, a_n \rangle$. In the latter case, however, b is a valid behaviour of P as well because the set of divergences is postfix-closed.

We now present the construction of the process $B(b, \mathcal{N})$ from step 1 above. $B(b, \mathcal{N}) = \langle S_B, \mathsf{init}_B, \Sigma_B, T_B \rangle$, where (see Figure 5.5):

- $S_B = \{q_0, \ldots, q_n\}$ and $\mathsf{init}_B = q_0$ and $\Sigma_B = \Sigma$,
- $T_B = \{q_i \xrightarrow{a_i} q_{i+1} \mid 0 \le i \le n-1\} \cup \{q_n \xrightarrow{a_n} q_r\}$,
- $\mathsf{min\_acceptances}(q_i) = \{\{a_i\}\}$,
- $\delta(q_i) = \text{false}$

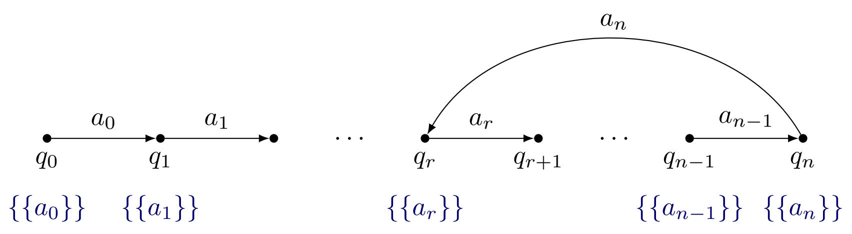


Figure 5.5: $B(b, \mathcal{N})$ for b bad lasso trace $\langle a_0, \ldots, a_{r-1}\rangle^\frown\langle a_r, \ldots, a_n\rangle^\omega$

5.5 Abstraction Refinement

Let us suppose that at some iteration of the CEGAR loop we establish that a given behaviour b introduced by an abstract execution ρ of the abstracted leaf $A = \langle S_A, \mathsf{init}_A, \Sigma, T_A\rangle$ is spurious for the original leaf $P = \langle S, \mathsf{init}, \Sigma, T\rangle$. Let us further suppose that $A = P/\!\sim$, i.e., that the abstraction A is induced by a given equivalence relation (partition) $\sim$ on S, with the sets of minimal acceptances and the divergence flags adjusted accordingly, depending on the CSP model $\mathcal{M}$.

In this section we present strategies for obtaining a proper refinement $\sim'$ of $\sim$ such that the corresponding abstraction $A' = P/\!\sim'$ does not allow the spurious execution ρ and maintains the invariant $A \sqsubseteq_{\mathcal{M}} A' \sqsubseteq_{\mathcal{M}} P$. We remark that after eliminating ρ from A, the resulting refinement A' might still have other instances of the spurious behaviour b manifested on other paths of the transition system. Methods for obtaining and eliminating all executions instances of b in A using subsequent refinement checks in FDR are presented as Algorithms 6 and 7, for finite and lasso behaviours, respectively. The constructions of $N(b, \mathcal{M})$ and $B(b, \mathcal{N})$ are described in the previous section; they are the same as the ones we used for counterexample validation. The function purge takes as input an abstraction A along with an execution ρ of A giving rise to an observable behaviour b, and generates a new refined abstraction A' by splitting blocks in A along ρ.

In the remainder of this section we focus on eliminating a given spurious execution ρ from A that gives rise to an observable behaviour b. In other words, we present strategies for implementing the function purge depending on the type of behaviour of b. We note that, even if we purge just a single execution ρ from A, instead of all execution instances of b in A, our definition of $\sim'$ as a proper refinement of $\sim$ satisfies the progress requirement

Algorithm 6 Obtaining all spurious executions of finite behaviours

```
Input: An abstraction $A$ of $P$ and an execution $\rho$ of $A$ exhibiting a behaviour $b$ spurious for $P$ in
    a semantic model $\mathcal{M}$
Output: A refined abstraction $A'$ with $A \sqsubseteq_{\mathcal{M}} A' \sqsubseteq_{\mathcal{M}} P$ and no manifestation of $b$ in $A'$
1: $\mathsf{purge}(A, b, \rho) \longmapsto A'$
2: while $N(b, \mathcal{M}) \not\sqsubseteq_{\mathcal{M}} A'$ do
3:     obtain a counterexample execution $\rho'$
4:     // $\rho'$ necessarily exhibits $b$ as well
5:     $\mathsf{purge}(A', b, \rho') \longmapsto A''$
6:     $A' = A''$
7: end while
8: return $A'$
```

Algorithm 7 Obtaining all spurious executions of lasso traces

```
Input: An abstraction $A$ of $P$ and an execution $\rho$ of $A$ exhibiting a lasso trace $b$ spurious for $P$
Output: A refined abstraction $A'$ with $A \sqsubseteq_{FD} A' \sqsubseteq_{FD} P$ and no manifestation of $b$ in $A'$
1: $\mathsf{purge}(A, b, \rho) \longmapsto A'$
2: while $(B(b, \mathcal{N}) \underset{\Sigma}{\parallel} A') \setminus \Sigma$ diverges and the counterexample $\rho'$ is a lasso do
3:     // $\rho'$ necessarily exhibits $b$ as well
4:     $\mathsf{purge}(A', b, \rho') \longmapsto A''$
5:     $A' = A''$
6: end while
7: return $A'$
```

necessary for the termination of the CEGAR loop.

To formalise our ideas, let us introduce a few auxiliary definitions and notations, revolving around the notions of abstraction and concretisation [CC77]. Throughout this section we will use the small Greek letters α and β to denote the states of A. Furthermore, we will write $\longrightarrow$ instead of $\longrightarrow_A$ to denote transitions of the abstract model A when the context is unambiguous. First we define $\sim$-concretisation of an abstract state α:

$$\sim^{-1}(\alpha) = \{s \in S \mid s \in \alpha\},$$

We lift this definition to a concretisation of an abstract execution fragment. If ρ is of the form:

$$\rho = \alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \ldots \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n,$$

then the $\sim$-concretisation of ρ is given by:

$$\sim^{-1}(\rho) = \{s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_n \mid s_i \in \alpha_i \text{ and for } 0 \le i \le n-1,\ s_i \xrightarrow{a_i} s_{i+1}\ \}.$$

We also use the following notation. For ρ defined as above and $0 \leq i \leq n$, let $\rho \restriction i$ be the execution fragment obtained from the first i steps of ρ, i.e.:

$$\rho \restriction i = \alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \ldots \alpha_{i-1} \xrightarrow{a_{i-1}} \alpha_i.$$

5.5.1 Simulation of Abstract Executions

Let $\rho = \alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \ldots \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n$ with $\alpha_0 = [\mathsf{init}]^{\sim}$ be an execution of $A = P/\!\sim$. We give an algorithm Simulate based on [CGJ+00] that aims to compute $\sim^{-1}(\rho)$, i.e., the set of concrete executions of $P = \langle \mathsf{init}, \mathsf{Post} \rangle$ that correspond to ρ. At each step i we define a set of reachable concrete states R_i inductively as follows:

$$R_0 = \sim^{-1}(\alpha_0) \cap \{\mathsf{init}\} = \{\mathsf{init}\}$$
$$R_i = \mathsf{Post}(R_{i-1}, a_{i-1}) \cap \sim^{-1}(\alpha_i) \quad \text{for } i = 1, \ldots, n$$

We present the algorithm Simulate below. It simulates the trace $\langle a_0, \ldots, a_{n-1} \rangle$ on P by propagating it as further as possible along $\sim^{-1}(\rho)$ executions of P. The algorithm computes the maximum index i such that $\sim^{-1}(\rho \restriction i) \neq \emptyset$, i.e., the maximum index i satisfying $R_i \neq \emptyset$.

Algorithm 8 Simulate (Based on [CGJ+00])

```
 1: $R' = \emptyset$
 2: $R = \sim^{-1}(\alpha_0) \cap \{\mathsf{init}\} = \{\mathsf{init}\}$
 3: $i = 0$
 4: while $i < n$ and $R \neq \emptyset$ do
 5:     $i = i + 1$
 6:     $R' = R$
 7:     $R = \mathsf{Post}(R', a_{i-1}) \cap \sim^{-1}(\alpha_i)$
 8: end while
 9: if $R \neq \emptyset$ then
10:     return $\langle R, i \rangle$
11: else
12:     return $\langle R', i - 1 \rangle$
13: end if
```

If upon termination the algorithm returns the tuple $\langle R, i \rangle$ with $i = n$, then the abstract execution ρ of A corresponds to a genuine execution of P and we can conclude that the trace $\langle a_0, \ldots, a_{n-1} \rangle$ is a valid trace of P. If $i < n$, then we call the abstract state α_i the *pivot* state of ρ (after [HJMS02]). We also refer to a_i as the *bad event*. Intuitively, the transition $\alpha_i \xrightarrow{a_i} \alpha_{i+1}$ is the last transition on the shortest failure of $\sim^{-1}(\rho)$ in P. The pivot abstract state α_i can be partitioned to the following sets of concrete states:

- $\mathsf{dead\_end}(\alpha_i) = R \neq \emptyset$ are the states of P that are reachable from init through $\langle a_0, \ldots, a_{i-1}\rangle$ along $\sim$-concretisations of $\rho \upharpoonright i$. For every $s \in \mathsf{dead\_end}(\alpha_i)$ there is no transition $s \xrightarrow{a_i} s'$ in P for any $s' \in \alpha_{i+1}$.
- $\mathsf{bad}(\alpha_i) = \mathsf{Pre}(\alpha_{i+1}, a_i) \neq \emptyset$ are the states in α_i that are not reachable from init through $\langle a_0, \ldots, a_{i-1}\rangle$ along $\sim$-concretisations of $\rho \upharpoonright i$, but have a_i-successors in α_{i+1}. Those are the concrete states that contribute to the failure. $\mathsf{bad}(\alpha_i) \neq \emptyset$ due to the (minimal) existential nature of the abstract transition relation and the fact that $\alpha_i \xrightarrow{a_i} \alpha_{i+1}$ is a valid abstract transition.
- $\mathsf{dont\_care}(\alpha_i) = \alpha_i - (\mathsf{dead\_end}(\alpha_i) \cup \mathsf{bad}(\alpha_i))$ is the set of concrete states in α_i that are neither reachable from init through $\langle a_0, \ldots, a_{i-1}\rangle$ along $\sim$-concretisations of $\rho \upharpoonright i$, nor have a_i-successors in α_{i+1}.

We highlight the following three observations: 1) both $\mathsf{dead\_end}(\alpha_i)$ and $\mathsf{bad}(\alpha_i)$ are non-empty, 2) $\mathsf{dont\_care}(\alpha_i)$ may or may not be empty, and 3) all states in $\mathsf{dead\_end}(\alpha_i)$ and $\mathsf{dont\_care}(\alpha_i)$ may have a_i-successors in abstract states different from α_{i+1}.

We now present our abstraction-refinement strategies depending on the type of spurious behaviour, assuming that we work in the most general of all three models $\mathcal{N}$. We also remark that we perform the abstraction-refinement step after already having established that the behaviour is spurious.

5.5.2 Abstraction-Refinement Strategies for Spurious Finite Traces

Let us first consider an abstract execution ρ of $A = P/\!\sim$

$$\rho = \alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \alpha_2 \quad \cdots \quad \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n \xrightarrow{a_n} \alpha_{n+1},$$

giving rise to a finite trace $w = \langle a_0, \ldots, a_n\rangle$ spurious for P. Since w is spurious for P, we have $\mathsf{Reach}(P, \mathsf{init}, w) = \emptyset$. We use the algorithm $\mathsf{Simulate}$ on ρ to provide the failure transition $\alpha_i \xrightarrow{a_i} \alpha_{i+1}$ of ρ together with the sets $\mathsf{dead\_end}(\alpha_i)$, $\mathsf{bad}(\alpha_i)$ and $\mathsf{dont\_care}(\alpha_i)$, as described in Section 5.5.1.

We now describe a collection of refinement strategies that achieve the goal to purge the spurious execution ρ from A. In all of those we derive a proper refinement $\sim'$ of $\sim$ that agrees with $\sim$ on all abstract states different from α_i and splits α_i to two or more equivalence classes in a way that detaches $\mathsf{bad}(\alpha_i)$ from $\mathsf{dead\_end}(\alpha_i)$. We recall that both $\mathsf{bad}(\alpha_i)$ and $\mathsf{dead\_end}(\alpha_i)$ are non-empty. For strategies T3 to T7 below, we create a new equivalence class α_{i_j} only if the latter is non-empty.

T1 Split α_i to:

1. $\alpha_{i_1} = \mathsf{bad}(\alpha_i)$, and
2. $\alpha_{i_2} = \mathsf{dead\_end}(\alpha_i) \cup \mathsf{dont\_care}(\alpha_i)$.

This corresponds to the original bisimulation quotienting algorithms suggested in [KS83, PT87].

T2 Split α_i to [CCO+05]:

1. $\alpha_{i_1} = \mathsf{dead\_end}(\alpha_i)$, and
2. $\alpha_{i_2} = \mathsf{bad}(\alpha_i) \cup \mathsf{dont\_care}(\alpha_i)$.

T3 Split α_i to:

1. $\alpha_{i_1} = \mathsf{dead\_end}(\alpha_i)$,
2. $\alpha_{i_2} = \mathsf{bad}(\alpha_i)$, and
3. $\alpha_{i_3} = \mathsf{dont\_care}(\alpha_i)$.

T4 Split α_i to:

1. $\alpha_{i_1} = \mathsf{bad}(\alpha_i)$,
2. $\alpha_{i_2} = \mathsf{dont\_care}(\alpha_i)$,
3. $\alpha_{i_3} = \{s \in \mathsf{dead\_end}(\alpha_i) \mid \mathsf{Post}(s, a_i) \neq \emptyset\}$, and
4. $\alpha_{i_4} = \{s \in \mathsf{dead\_end}(\alpha_i) \mid \mathsf{Post}(s, a_i) = \emptyset\}$.

Here we recall that states in $\mathsf{dead\_end}(\alpha_i)$ may have a_i-successors to states not in α_{i+1}.

T5 Split α_i to :

1. $\alpha_{i_1} = \mathsf{dead\_end}(\alpha_i)$,

2. $\alpha_{i_2} = \mathsf{bad}(\alpha_i)$,
3. $\alpha_{i_3} = \{s \in \mathsf{dont\_care}(\alpha_i) \mid \mathsf{Post}(s, a_i) \neq \emptyset\}$, and
4. $\alpha_{i_4} = \{s \in \mathsf{dont\_care}(\alpha_i) \mid \mathsf{Post}(s, a_i) = \emptyset\}$.

T6 Split α_i to [Val09, VF10]:

1. $\alpha_{i_1} = \{s \in \mathsf{bad}(\alpha_i) \mid \mathsf{Post}(s, a_i) \subseteq \alpha_{i+1}\}$,
2. $\alpha_{i_2} = \{s \in \mathsf{bad}(\alpha_i) \mid \mathsf{Post}(s, a_i) \cap \alpha_{i+1} \neq \emptyset$ and $\mathsf{Post}(s, a_i) - \alpha_{i+1} \neq \emptyset\}$, and
3. $\alpha_{i_3} = \mathsf{dead\_end}(\alpha_i) \cup \mathsf{dont\_care}(\alpha_i)$.

T7 Split α_i to:

1. $\alpha_{i_1} = \{s \in \mathsf{bad}(\alpha_i) \mid \mathsf{Post}(s, a_i) \subseteq \alpha_{i+1}\}$,
2. $\alpha_{i_2} = \{s \in \mathsf{bad}(\alpha_i) \mid \mathsf{Post}(s, a_i) \cap \alpha_{i+1} \neq \emptyset$ and $\mathsf{Post}(s, a_i) - \alpha_{i+1} \neq \emptyset\}$,
3. $\alpha_{i_3} = \mathsf{dead\_end}(\alpha_i)$, and
4. $\alpha_{i_4} = \mathsf{dont\_care}(\alpha_i)$.

T8 Split α_i to multiple equivalence classes with respect to their abstract a_i-successors [COYC03]. For any $s, q \in \alpha_i$, $s \sim' q$ if and only if $\{[s']^{\sim} \mid s \xrightarrow{a_i} s'\} = \{[q']^{\sim} \mid q \xrightarrow{a_i} q'\}$, i.e., if s and q have overlapping sets of abstract a_i-successors with respect to $\sim$. For any $s \notin \alpha_i$, $[s]^{\sim'} = [s]^{\sim}$.

5.5.3 Abstraction-Refinement Strategies for Spurious Minimal Acceptances

Let us now consider the case when the spurious behaviour b is an abstract execution ρ of $A = P/\!\sim$:

$$\rho = \alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \alpha_2 \quad \cdots \quad \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n,$$

giving rise to a bad minimal-acceptance set $B \subseteq \Sigma$ at state α_n. The spuriousness of b can manifest itself in two ways:

1. either ρ cannot be matched by a valid execution of P, i.e., $\sim^{-1}(\rho) = \emptyset$,
2. or, for all valid executions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \quad \cdots \quad s_{n-1} \xrightarrow{a_{n-1}} s_n \in \sim^{-1}(\rho)$, the set of events B is not an acceptance of s_n, i.e., for all $C \in \mathsf{min\_acceptances}(s_n)$, $C \nsubseteq B$.

Whether ρ is spurious because of reason 1 or 2 above we can check by running the algorithm $\mathsf{Simulate}$ on ρ. Let us suppose that $\mathsf{Simulate}$ returns the tuple of values $\langle R, i\rangle$ upon termination. As described in Section 5.5.1, if $i < n$, then ρ cannot be concretised to a valid execution of P and we use the abstraction-refinement strategies described in Section 5.5.2.

Let us now focus on the case when $\mathsf{Simulate}$ returns $i = n$, together with the set of reachable states $R \subseteq \alpha_n$ of P. We distinguish different cases depending on the type of abstraction that we use for minimal acceptances (see Section 5.3.3).

Proposal 1. Let us first consider the case when for each abstract state α we let:

$$\mathsf{min\_acceptances}(\alpha) = \mathsf{min\_basis}(\bigcup_{s \in \alpha} \mathsf{min\_acceptances}(s)).$$

Then, since $B \in \mathsf{min\_acceptances}(\alpha_n)$, by construction there exists $s_b \in \alpha_n$, such that B is an acceptance of s_b. For the sake of the argument, let us suppose that $s_b \in R$. Then there would be a valid execution $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \quad \cdots \quad s_{n-1} \xrightarrow{a_{n-1}} s_n$ of P such that $s_n = s_b$ and B is an acceptance of s_n. But then the behaviour b would not be spurious. Therefore $s_b \notin R$. We distinguish three types of concrete states in α_n:

1. $\mathsf{dead\_end}(\alpha_n) = R \neq \emptyset$ is the set of concrete states of P that are reachable from init through $\langle a_0, \ldots, a_{n-1}\rangle$ along $\sim$-concretisations of ρ. For every $s \in \mathsf{dead\_end}(\alpha_n)$, B is not an acceptance of s. $\mathsf{dead\_end}(\alpha_n) \neq \emptyset$ because the algorithm $\mathsf{Simulate}$ returns an index $i = n$.

2. $\mathsf{bad}(\alpha_n) = \{s \in \alpha_n \mid B \text{ is an acceptance (not necessarily minimal) of } s\}$. $\mathsf{bad}(\alpha_n) \neq \emptyset$ because B is a minimal acceptance of α_n. Those are the concrete states that contribute to b being spurious.

3. $\mathsf{dont\_care}(\alpha_n) = \alpha_n - (\mathsf{dead\_end}(\alpha_n) \cup \mathsf{bad}(\alpha_n))$, which may or may not be empty.

In order to eliminate the spurious behaviour b, we introduce a proper refinement $\sim'$ of $\sim$ that agrees on all abstract states different from α_n and separates the two non-empty sets $\mathsf{dead\_end}(\alpha_n)$ and $\mathsf{bad}(\alpha_n)$ of α_n in different equivalence classes under $\sim'$. Any of the following strategies below achieves this objective and A13 is only possible if $\mathsf{dont\_care}(\alpha_n) \neq \emptyset$.

A11 Split α_n to $\alpha_{n_1} = \mathsf{bad}(\alpha_n)$ and $\alpha_{n_2} = \mathsf{dead\_end}(\alpha_n) \cup \mathsf{dont\_care}(\alpha_n)$.

A12 Split α_n to $\alpha_{n_1} = \mathsf{dead\_end}(\alpha_n)$ and $\alpha_{n_2} = \mathsf{bad}(\alpha_n) \cup \mathsf{dont\_care}(\alpha_n)$.

A13 Split α_n to $\alpha_{n_1} = \mathsf{dead\_end}(\alpha_n)$, $\alpha_{n_2} = \mathsf{bad}(\alpha_n)$ and $\alpha_{n_3} = \mathsf{dont\_care}(\alpha_n)$.

Proposal 2. Let us now consider the case when for each abstract state α we let:

$$\mathsf{min\_acceptances}(\alpha) = \begin{cases} \mathsf{min\_acceptances}(s) & \text{if } \alpha \text{ is the singleton set } \{s\} \\ \{\bigcap_{s\in\alpha} \bigcap_{A_k \in \mathsf{min\_acceptances}(s)} A_k\} & \text{otherwise} \end{cases}$$

Given the set of concrete states $R \subseteq \alpha_n$ of P, reachable after $\langle a_0, \ldots, a_{n-1}\rangle$ along $\sim$-concretisations of ρ, we first check whether $\alpha_n - R \neq \emptyset$. If this is the case, we use the following refinement $\sim'$ of $\sim$, which agrees with $\sim$ outside α_n.

A21 Split α_n to $\alpha_{n_1} = R$ and $\alpha_{n_2} = \alpha_n - R$.

If all states of α_n are reachable, i.e., $\alpha_n = R$, then we claim that α_n consists of at least two concrete states. Indeed, suppose for the sake of the argument that the abstract state α_n, which exhibits the bad minimal acceptance B, is the singleton set $\{s\}$ for some $s \in S$. But then by construction $\mathsf{min\_acceptances}(\alpha_n) = \mathsf{min\_acceptances}(s)$, whereby B would be a minimal acceptance of s. Then, since $s \in R$, the behaviour b would not be spurious for P. Therefore, $\alpha_n = R$ contains at least two states s_{n_1} and s_{n_1}. We detach s_{n_1} from α_n in $\sim'$ and let $\sim'$ of $\sim$ agree on all equivalence classes different from α_n.

A22 Split α_n to $\alpha_{n_1} = \{s_{n_1}\}$ and $\alpha_{n_2} = \alpha_n - \{s_{n_1}\}$.

As a result, all minimal acceptances of s_{n_1} will be expanded in the abstract state $[s_{n_1}]^{\sim'}$.

Proposal 3. Finally we consider the case when for each abstract state α we let:

$$\mathsf{min\_acceptances}(\alpha) = \begin{cases} \mathsf{min\_acceptances}(s) & \text{if } \alpha \text{ is the singleton set } \{s\} \\ \mathsf{min\_acceptances}(s_k) & \text{if } |\,\alpha\,| > 1 \text{ and } \exists\, s_k \in \alpha\,\centerdot \\ & \forall\, s \in \alpha\,\centerdot\, s_k \sqsubseteq_f s \\ \{\bigcap_{s\in\alpha} \bigcap_{A_k \in \mathsf{min\_acceptances}(s)} A_k\} & \text{otherwise} \end{cases}$$

This case is handled very similarly to the previous one. Given the set of concrete states $R \subseteq \alpha_n$ of P, reachable after $\langle a_0, \ldots, a_{n-1}\rangle$ along $\sim$-concretisations of ρ, we first check whether $\alpha_n - R \neq \emptyset$. If so, we use the same refinement as $A21$.

A31 Split α_n to $\alpha_{n_1} = R$ and $\alpha_{n_2} = \alpha_n - R$.

If $\alpha_n = R$, again α_n contains at least two concrete states s_{n_1} and s_{n_1}, so we can just move one of them in a new equivalence class under $\sim'$.

A32 Split α_n to $\alpha_{n_1} = \{s_{n_1}\}$ and $\alpha_{n_2} = \alpha_n - \{s_{n_1}\}$.

With this abstraction, however, we can as well try to identify a concrete state $s_k \in \alpha_n$ and a subset Q of α_n such that $Q = \{s \in \alpha_n \mid s_k \sqsubseteq_f s\}$. Q will be a proper subset of α_n because otherwise the behaviour b would have been spurious for the concrete execution ending in s_k.

A33 Split α_n to $\alpha_{n_1} = Q$ and $\alpha_{n_2} = \alpha_n - Q$.

In this case the minimal acceptances of Q under $\sim'$ will be fully expanded to the set $\mathsf{min\_acceptances}(s_k)$.

5.5.4 Abstraction-Refinement Strategies for Spurious Divergences

Let us now consider the case when the spurious behaviour b is an abstract execution ρ of $A = P/{\sim}$

$$\rho = \alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \alpha_2 \quad \cdots \quad \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n,$$

ending in a divergent abstract state α_n. The reasons for b being spurious can be the following:

1. either ρ does not concretise to a valid execution of P, i.e., $\sim^{-1}(\rho) = \emptyset$,

2. or, for all valid executions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \quad \cdots \quad s_{n-1} \xrightarrow{a_{n-1}} s_n \in \sim^{-1}(\rho)$, the last concrete state s_n is not divergent.

Whether ρ is spurious because of reason 1 or 2 above we can check by running the algorithm $\mathsf{Simulate}$ on ρ. Let us suppose that $\mathsf{Simulate}$ returns the tuple of values $\langle R, i\rangle$ upon termination. As described in Section 5.5.1, if $i < n$, then ρ cannot be concretised to a valid execution of P and we use the abstraction-refinement strategies described in Section 5.5.2.

Let us now focus on the case when $\mathsf{Simulate}$ returns $i = n$, together with the non-empty set of reachable states $R \subseteq \alpha_n$ of P. It is clear then that all concrete states in R are not divergent as otherwise the behaviour b would not have been spurious. Let us notice as well that, because α_n is divergent, there exists a concrete state $s_b \in \alpha_n$ which is divergent. Therefore we identify three types of concrete states in α_n:

1. $\mathsf{dead\_end}(\alpha_n) = R \neq \emptyset$ is the set of concrete states of P that are reachable from init through $\langle a_0, \ldots, a_{n-1}\rangle$ along $\sim$-concretisations of ρ. For every $s \in \mathsf{dead\_end}(\alpha_n)$, s is not divergent.

2. $\mathsf{bad}(\alpha_n) = \{s \in \alpha_n \mid s \text{ is divergent}\}$. $\mathsf{bad}(\alpha_n) \neq \emptyset$ because α_n is divergent. Those are the concrete states that contribute to b being spurious.

3. $\mathsf{dont\_care}(\alpha_n) = \alpha_n - (\mathsf{dead\_end}(\alpha_n) \cup \mathsf{bad}(\alpha_n))$, which may or may not be empty.

Then in the new partition $\sim'$ we can employ the following strategies for splitting α_n (and for $s \notin \alpha_n$,$[s]^{\sim'} = [s]^{\sim}$):

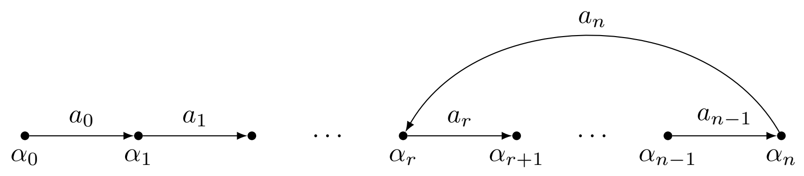
D1 Split α_n to $\alpha_{n_1} = \mathsf{bad}(\alpha_n)$ and $\alpha_{n_2} = \mathsf{dead\_end}(\alpha_n) \cup \mathsf{dont\_care}(\alpha_n)$.

D2 Split α_n to $\alpha_{n_1} = \mathsf{dead\_end}(\alpha_n)$ and $\alpha_{n_2} = \mathsf{bad}(\alpha_n) \cup \mathsf{dont\_care}(\alpha_n)$.

D3 Split α_n to $\alpha_{n_1} = \mathsf{dead\_end}(\alpha_n)$, $\alpha_{n_2} = \mathsf{bad}(\alpha_n)$ and $\alpha_{n_3} = \mathsf{dont\_care}(\alpha_n)$.

5.5.5 Abstraction-Refinement Strategies for Spurious Lasso Traces

Let us finally consider the case when the spurious behaviour b is an infinite word $w = \langle a_0, \ldots, a_{r-1}\rangle^\frown\langle a_r, \ldots, a_n\rangle^\omega$ resulting from an abstract execution ρ as depicted in Figure 5.6 (where $0 \leq r \leq n$).



$$\begin{tikzcd}
\alpha_0 \arrow[r, "a_0"] & \alpha_1 \arrow[r, "a_1"] & \bullet & \cdots & \alpha_r \arrow[r, "a_r"] & \alpha_{r+1} & \cdots & \alpha_{n-1} \arrow[r, "a_{n-1}"] & \alpha_n \arrow[llll, bend right, "a_n"']
\end{tikzcd}$$

Figure 5.6: A spurious lasso trace

We first simulate the trace $t = \langle a_0, \ldots, a_{r-1}, a_r, \ldots, a_n\rangle$ on P using the algorithm $\mathsf{Simulate}$ to obtain the tuple of return values $\langle R, i\rangle$. If $i < n + 1$, then

$$\sim^{-1}(\alpha_0 \xrightarrow{a_0} \alpha_1 \xrightarrow{a_1} \ldots \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n \xrightarrow{a_n} \alpha_{n+1} = \alpha_r) = \emptyset$$

and we use the abstraction-refinement strategies described in Section 5.5.2.

If $i = n + 1$, we check whether $\alpha_{n+1} = \alpha_r = R$. If not, we can just detach R from α_r to obtain a proper refinement $\sim'$ of $\sim$. For any $s \notin \alpha_r$, we let $[s]^{\sim'} = [s]^{\sim}$.

L1 Split α_r to $\alpha_{r_1} = R$ and $\alpha_{r_2} = \alpha_r - R$.

If $i = n + 1$, regardless of whether $R = \alpha_r$ or not, we can use the algorithm $\mathsf{Simulate}$ on ρ until it terminates, i.e., until at some iteration of the loop $\alpha_r \xrightarrow{a_r} \alpha_{r+1} \ldots \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n \xrightarrow{a_n} \alpha_r$, $\mathsf{Simulate}$ reaches a set of dead-end states $R' \subseteq \alpha_j$ for some $j \in \{\alpha_r, \ldots, \alpha_n\}$ and $\mathsf{Post}(R', a_j) \cap \alpha_{j+1} = \emptyset$ (we assume $\alpha_{n+1} = \alpha_r$). Because ρ is spurious, $\mathsf{Simulate}$ reaches such a tuple $\langle R', j\rangle$ after at most $\min_{r \leq k \leq n} |\, \alpha_k \,|$ iterations of the loop $\alpha_r \xrightarrow{a_r} \alpha_{r+1} \ldots \alpha_{n-1} \xrightarrow{a_{n-1}} \alpha_n \xrightarrow{a_n} \alpha_r$, as proven in [CGJ$^+$00, CGJ$^+$03]. Having the tuple $\langle R', j\rangle$, we then use the abstraction refinement strategies described in Section 5.5.2 to obtain a proper refinement $\sim'$ of $\sim$.

L2 Keep simulating the infinite abstract execution ρ until reaching a pivot state α_j. Split α_j according to the selected strategy for refining abstract finite traces presented in Section 5.5.2.

5.6 Implementation Details

We have implemented the CEGAR framework presented in this chapter in a prototype tool developed in C++ on top of FDR.

As we pointed out in Section 5.3.3, abstracting models based on traces and abstracting models based on stable failures are triggered by two opposite forces: the coarser the partitions on the state spaces of the leaf processes, the better the performance in the traces model and the worse in the stable-failures model (and conversely). In order to achieve a sensible balance between those two forces, our CEGAR framework performs two successive CEGAR loops if working in the stable-failures or the failures-divergences model of CSP. The first CEGAR loop focuses on establishing or refuting the trace-refinement relationship exclusively. If trace refinement holds, the second CEGAR loop takes as initial partitions the partitions obtained by the trace CEGAR loop and continues refining them as in the standard CEGAR framework.

5.6.1 Aggressive vs. Lazy Strategies

We have implemented and experimented with different strategies regarding the four main stages of the CEGAR loop. More aggressive strategies generally yield a shorter sequence of abstractions with larger state spaces. Lazy strategies refine abstractions only when necessary, aiming to generate coarser partitions. In this case, the subsequent abstractions have smaller state spaces, but the CEGAR loop might take a greater number of iterations to converge. The choice of a particular strategy is configurable and some of the aggressive strategies remain for future work. Nonetheless, empirical evidence suggests that generally lazy strategies are more successful in combating the state-space explosion problem and, hence, yield better performance results.

Regarding abstraction, our prototype supports all abstraction schemes described in Section 5.3, including all three proposals for abstracting minimal acceptances. Regarding initial partitions, the laziest strategy corresponds to aggregating together all states of a leaf process. A slightly more precise initial abstraction if working in the failures-divergences model is to partition the concrete state space into two equivalence classes—one for all immediately divergent states and one for all immediately non-divergent states (provided both classes are non-empty). An aggressive strategy for initial abstraction would be to aggregate together all states that are willing to communicate exactly the same sets of visible events, although we have not implemented this strategy yet.

In the refinement checking phase, at each iteration of the CEGAR loop we can consider a single counterexample behaviour (of the entire implementation process) or multiple ones. The latter one is a valid option as FDR provides functionality for extending the state-space traversal until a configurable number of counterexample executions are found. Again, our prototype tool currently supports the former lazy option only.

In the abstraction-refinement phase, we have different options for the number of leaf processes to refine, the number of executions per spurious behaviour to eliminate and the strategy to follow when splitting abstract states.

Firstly, we can lazily refine a single process [COYC03, CCO$^+$05] or aggressively refine all leaf processes whose contribution behaviours are spurious. Our prototype tool supports both options and we consider implementing a third option that randomises the number of leaves to refine.

Regarding the number of executions giving rise to a spurious behaviour in a given abstract leaf process, we have the option to eliminate just a single execution (the one reported by FDR), all possible executions, or a random number of those. Currently only the laziest first option is implemented, although in Section 5.5 we have demonstrated how to get hold of subsequent execution instances of a spurious behaviour with the help of FDR.

And finally, our tool supports a variety of strategies for splitting abstract states, depending on the semantic model and the type of spurious behaviour detected. Those currently include all strategies presented in Section 5.5, apart from T8, A33 and L2.

5.7 Experimental Results

In this section, we analyse the performance of our prototype tool on a number of case studies, comparing it against the performance of FDR 2.91. We experimented with verifying both safety and liveness properties, including checks for livelock and deadlock. Our test cases include Milner's scheduler, the alternating bit protocol, the mad postman network routing protocol [YJ89], a distributed database algorithm [Ros98, Ros90], Fischer's mutual exclusion algorithm and the dining philosophers.

The experimental results are summarised in Tables 5.1, 5.2 and 5.3 below. In all tables, times are reported in seconds, with * denoting a 30-minute timeout. The columns titled $\sharp$ report the number of iterations that it takes for CEGAR to converge, and the columns titled Refinement reflect the abstraction-refinement strategies that are used for the particular test case. All experimental results use Proposal 3 for abstracting minimal acceptances. The experiments were performed on a 3.07 GHz Intel Xeon processor with 8 GB RAM running Linux Ubuntu.

We note that the column for FDR represents its use with none of its compression functions used. As shown in [Ros98], applying compression techniques on some of those systems (e.g., on Milner's scheduler) results in a significant state-space reduction and, hence, in a much better performance. However, in general, picking the right compression function and compression strategy often requires a lot of experience and skill: it can be highly beneficial, detrimental, or not have any effect at all.

In terms of performance, experiments indicated that the CEGAR approach can either significantly boost or significantly deteriorate the process of refinement checking, depending on whether or not the property is established or refuted before all leaf processes get fully expanded. We observed such polar results when verifying both safety and liveness properties, as illustrated in Tables 5.1 and 5.2.

Table 5.1: CEGAR framework: experimental results. Times reported are in seconds, with * denoting a 30-minute timeout.

Property	**Benchmark**	**FDR**	**CEGAR**	$\sharp$	**Refinement**
Trace	Milner-10	0	0.03	21	T1
(safety),	Milner-20	158	0.07	41	
holds	Milner-30	*	0.16	61	
	Milner-50	*	0.64	101	
	Milner-100	*	4.42	201	
	Milner-200	*	40.01	401	
Deadlock	Mad Postman-3	4	0.03	4	T1, A31 + A32
(liveness),	Mad Postman-5	*	0.22	4	
holds	Mad Postman-7	*	1.49	4	
	Mad Postman-9	*	7.13	4	
Livelock	Mad Postman-3	6	0.04	11	T1, A31 + A32 + D2
(liveness),	Mad Postman-4	*	4.52	18	
holds	Mad Postman-5	*	*	—	
Buffer property (liveness), violated	Divergent ABP-5	0	0.2	20	T1, A31 + A32 + D2
Mutual exclusion (safety), holds	Fischer-5-3-20	0	29.3	22	T5 + refine multiple leaves
Livelock	DistributedDB-5		2.73	310	T1, A31 + A32 + D2
(liveness),	DistributedDB-6	15	26.1	515	
holds	DistributedDB-7	541	1353.02	780	
	DistributedDB-8	*	*	—	

For Milner's scheduler, the a-rotation property (as described in Section 2.2.2) is established after all cell process are refined to contain three instead of four states. The CEGAR loop using abstraction-refinement strategy T1 is presented in Figures 5.7 and 5.8, where the underlined concrete states are the bad states at pivot abstract states and spurious leaf traces are depicted in red. If considering just a single leaf process, the abstraction yields a reduction of only 25% in its state space. However, due to the state-space explosion phenomenon, when considering networks of multiple processes composed in parallel, even such a minor reduction yields an enormous boost in the performance of the entire network. The CEGAR framework outperforms the standard single-pass exploration mode of FDR by orders of magnitude, even for networks of relatively average size of 20 or 30, as recorded in Table 5.1.

The mad postman [YJ89] is a deadlock-free routing algorithm for forwarding message packets across a network. It assumes a rectangular grid of network points and supports message transfer from any source point to any destination point, where points can directly communicate with their immediate neighbours only. In our experiments, a mad postman network of size n constitutes a square grid of node processes $\{N_{i,j} \mid 1 \leq i, j \leq n\}$, where each node process $N_{i,j}$ can be concisely and elegantly modelled as a parallel composition of two processes $I_{i,j}$ and $O_{i,j}$. The key idea is that $N_{i,j}$ delegates different responsibilities to its component processes. $I_{i,j}$ can only intercept messages from a user, forward messages to the right and down I neighbours $I_{i+1,j}$ and $I_{i,j-1}$, respectively, as well as transfer a message to $O_{i,j}$. On the other hand, $O_{i,j}$ is responsible for advancing messages to the left and up neighbours $O_{i-1,j}$ and $O_{i,j+1}$, as well as for delivering messages to recipient users. In a nutshell, a message intended from a user at $N_{i,j}$ to a user at $N_{k,l}$ is input by $I_{i,j}$, propagated as far to the right and down as necessary until reaching a transfer point $I_{n,m}$, passed from $I_{n,m}$ to $O_{n,m}$, then propagated from $O_{n,m}$ to $O_{k,l}$ and then subsequently to the recipient user. The values for n and m are calculated as $n = \max(i, k)$ and $m = \min(j, l)$ and a CSP modelling of the entire algorithm can be found in [Ros98].

An interesting observation is that establishing both deadlock and livelock freedom of the mad postman algorithm is in a way data independent. Deadlock freedom is established when for some $1 \leq i, j \leq n$, $I_{i,j}$ and $O_{i,j}$ get fully expanded, regardless of the size of the grid. The first iteration is used to establish that the property holds in the traces model. Iterations two and three expand $I_{i,j}$ and $O_{i,j}$, respectively, and iteration four proves sufficient to establish that the property holds in the stable-failures model. Similarly, livelock freedom for the mad postman protocol is established after all $I_{i,j}$ get fully expanded, again regardless of the size of the grid. For verifying both deadlock and livelock freedom, the CEGAR framework significantly outperforms FDR, as is apparent from the results in Table 5.1.

For the alternating bit protocol with media losing up to 5 messages in a row (see Section 4.2), we check whether the buffer property holds for the unconstrained divergent version. The property is proven violated after reducing the state spaces of the erroneous media $E(4)$ and $F(4)$ from 21 to 5 states and from 11 to 5 states, respectively. For all livelock-free versions of ABP (for which the buffer property holds) there is no reduction in the state space.

Similarly for Fischer's mutual exclusion protocol for 5 processes using time constants 3 and 20, we detect a fair amount of state-space reduction, which, however, proves insufficient for outperforming FDR.

The experiments with the distributed database illustrate how employing CEGAR can also significantly worsen the refinement check, especially if working in CSP models that take account of acceptances or refusals. As pointed out in Section 4.9.2, establishing livelock freedom of the distributed database is exceptionally intricate and the CEGAR approach yielded no reduction at all. Consequently, FDR is considerably more efficient in establishing livelock freedom.

In Table 5.2, we analyse the performance of the CEGAR framework on a collection of implementations of the classical dining philosophers—the original one that deadlocks, an asymmetric one that includes a single left-handed philosopher, as well as a version that employs a butler to schedule seats. As opposed to the distributed database, establishing livelock freedom of this class of scripts is trivial and is accomplished by our prototype in just two iterations, tremendously outperforming FDR. As it comes to establishing deadlock freedom, however, the performance of the CEGAR prototype deteriorates compared to that of FDR, despite achieving reductions in the state space. For both the deadlock-free and the deadlock-manifesting versions, the prototype converges before all leaf processes get fully expanded, but records a substantial overhead in intermediate stable-failures checks due to the blow-up of minimal acceptances.

In Table 5.3, we compare a number of strategies for refining abstractions when encountering spurious traces, as defined in Section 5.3.2. The experiment is performed in the context of establishing livelock freedom of the distributed database algorithm [Ros98, Ros90]

Table 5.2: CEGAR framework: experimental results for the dining philosophers. The strategies for abstraction refinement used are T1, A31 + A32 + D2.

		Dining Philosophers								
Property	**N**	**Standard**			**Asymmetric**			**Butler**		
		FDR	**CEGAR**	$\sharp$	**FDR**	**CEGAR**	$\sharp$	**FDR**	**CEGAR**	$\sharp$
Livelock (liveness), holds	5	0	0	2	0	0	2	0	0	2
	6	0	0.01	2	0	0.01	2	0	0.01	2
	7	1	0.01	2	1	0.01	2	1	0.01	2
	8	11	0.01	2	11	0.01	2	9	0.01	2
	9	93	0.02	2	90	0.02	2	76	0.02	2
	10	846	0.02	2	824	0.02	2	756	0.02	2
	50	—	1.95	2	—	2.06	2	—	2.22	2
Deadlock (liveness), holds for Asymmetric and Butler	5	0	0.25	28	0	0.40	32	0	0.15	46
	6	0	1.64	34	0	6.58	41	0	0.87	55
	7	0	15.34	38	0	34.41	43	1	5.65	64
	8	2	144.65	44	16	82.90	50	14	47.69	73
	9	15	834.05	49	121	720	47	105	386.72	81
	10	101	—	—	864	—	—	807	—	—

for 6 nodes. For each of the refinement strategies, we also compare the performance and convergence rate of the CEGAR framework depending on how many leaf processes are refined on each iteration. In this respect, $c = 1$ indicates refining a single leaf process per iteration, and $c = *$ indicates refining all leaf processes that contribute spurious behaviours. Let us observe the following:

- When refining a single leaf, the strategies that do not mix bad and don't care states typically yield better results (e.g., T1, T4 and T6). Out of those strategies, the most conservative one (T1, which detaches the bad states only) performs the best, even though it takes the greatest number of iterations to converge. In terms of performance, the worst of all strategies is T5, the one that undertakes the least number of CEGAR cycles. This is the strategy that further partitions don't care states. However, the conclusions can not be generalised: the counterexample is strategy T2 that does not mix bad and don't care states but yields a second worst result in terms of performance.

- If comparing each strategy in terms of number of leaves refined at each iteration of the CEGAR loop, there is no general rule either. For all of them, multiple leaf refinement makes convergence faster, but for some strategies (T3, T5, T7) the performance is

Table 5.3: CEGAR framework: comparison of different strategies for trace refinement. The test case is establishing livelock freedom of a distributed database with 6 nodes. The rows titled $c = 1$ and $c = *$ denote the number of leaves to refine at each iteration, with 1 meaning one and $*$ meaning all spurious component processes (but each leaf process at most once).

DistributedDB-6							
	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**
$c = 1$	24.88	41.51	34.63	26.83	97.59	23.09	27.79
$\sharp$	514	345	268	263	220	523	276
$c = *$	22.71	41.09	18.67	24.48	32.6	35.53	21.24
$\sharp$	240	302	151	128	120	191	141

significantly better, for others it is comparable (T1, T2, T4), and for T6 it is worse.

Based on those observations, we speculate that perhaps better results can be obtained if, at each iteration, we randomise the choice of refinement strategy and number of leaves to refine.

5.8 Conclusion and Future Work

In this chapter, we introduced a fully automated and compositional CEGAR framework for refinement checking in the traces, stable-failures and failures-divergences models of CSP. We proposed strategies for carrying out the stages of initial abstraction, counterexample validation and abstraction refinement component-wisely on leaf-process level, exploiting theoretical results on the compositionality, transitivity and monotonicity of CSP operators and facilitated by the capabilities of FDR. To the best of our knowledge, our work constituted the first application of CEGAR, in its automated form, to the setting of CSP.

Experiments with a prototype tool developed on top of FDR indicated that the CEGAR approach was in a position to significantly boost the performance of FDR when verifying correctness properties which did not address the full spectrum of details present in the system. We witnessed a significant enhancement in terms of performance when verifying both safety and liveness properties, including checks for livelock and deadlock. High performance, however, depended on whether or not the specification property was established or refuted before all leaves of the implementation process got fully expanded. In the negative, the subsequent abstraction/refinement steps naturally incurred a large performance overhead.

A certain limitation of our approach stems from the fact that abstracting models based on traces and abstracting models based on stable failures are triggered by two opposite forces. Despite adopting abstraction/refinement strategies for balancing those forces and an algorithm employing two consecutive CEGAR loops, we sometimes observed a massive blow-up of the minimal acceptance sets in intermediate refinement steps. In order to further minimise abstract minimal acceptances, we plan to investigate methods and possibly employ symbolic (or antichain) techniques for efficiently identifying structural refinements between groups of concrete states. Furthermore, we intend to make the refinement strategies more flexible and let them split abstract states into multiple parts on demand, should suitable structural refinements be detected.

As an alternative, we plan to extend the diamond elimination compression technique [RGG$^+$95, Ros11b], that we used as a preprocessing step, by introducing further methods for detecting pairs of LTS states P, Q satisfying $P \sqsubseteq Q$. After diamond elimination, in contrast to after normalisation, there can be ambiguous branching with respect to visible events. The technique is grounded on the observation that if $N \xrightarrow{a} P$, $N \xrightarrow{a} Q$ and $Q \in \tau^*(P)$, then $P \sqsubseteq Q$ and eliminating Q would not change the semantic value of the process. In the current implementation of FDR, iterated τ reachability is the only method for detecting such structural refinement of states. We plan to introduce other techniques based on simulation relations. For example, if P weakly or divergence-respecting weakly simulates Q, then $P \sqsubseteq Q$ also holds and we can safely remove Q.

Different classes of simulation relations can be further exploited to obtain an equivalence relation which is weaker than bisimulation but stronger than trace equivalence [vG00, BK08]. As defined in [BK08], P simulates Q if there exists a simulation relation R with $(P, Q) \in R$. P is then simulation equivalent to Q if there exist simulation relations R_1 and R_2 such that $(P, Q) \in R_1$ and $(Q, P) \in R_2$. As R_1^{-1} need not be the same as R_2, simulation equivalence is weaker than bisimulation and will result in a greater state-space reduction. Furthermore, algorithms for simulation quotienting [BK08] can be embedded in the CEGAR framework described in this chapter.

Other immediate extensions of our CEGAR framework include developing heuristics for deciding how many leaf processes to refine at each step, how many spurious executions of a process to eliminate and which refinement strategy to apply. We speculate that also introducing a certain degree of randomness in these decision processes would produce consistently more favourable results, as is the case with state-of-the-art SAT solvers such as MiniSAT.

We speculate that for systems that exhibit any form of symmetry, we can purge *classes* of counterexamples, instead of just a single counterexample, by applying a certain refinement step on all symmetric leaf processes instead of on just one. As illustrated with the CEGAR loop for Milner's scheduler (see Figures 5.7 and 5.8), the symmetric leaf processes *Cell*(*1*) and *Cell*(*2*) end up exhibiting the very same spurious counterexamples and undergoing the very same abstraction-refinement steps. Hence, after identifying the symmetry between *Cell*(*1*) and *Cell*(*2*), we could apply any given abstraction refinement of *Cell*(*1*) also on *Cell*(*2*), and vice versa, within a single CEGAR iteration, thus speeding up the convergence of the CEGAR loop.

As more general extensions of the CEGAR framework, we plan to exploit further abstraction mechanisms (e.g., counter abstraction, data abstraction, context-bounded analysis) in the context of CEGAR and/or its mirror opposite—underapproximation widening [GLST05]. We intend to also investigate methods for combining CEGAR with assume-guarantee reasoning along the lines of [CCGP, BPG08].

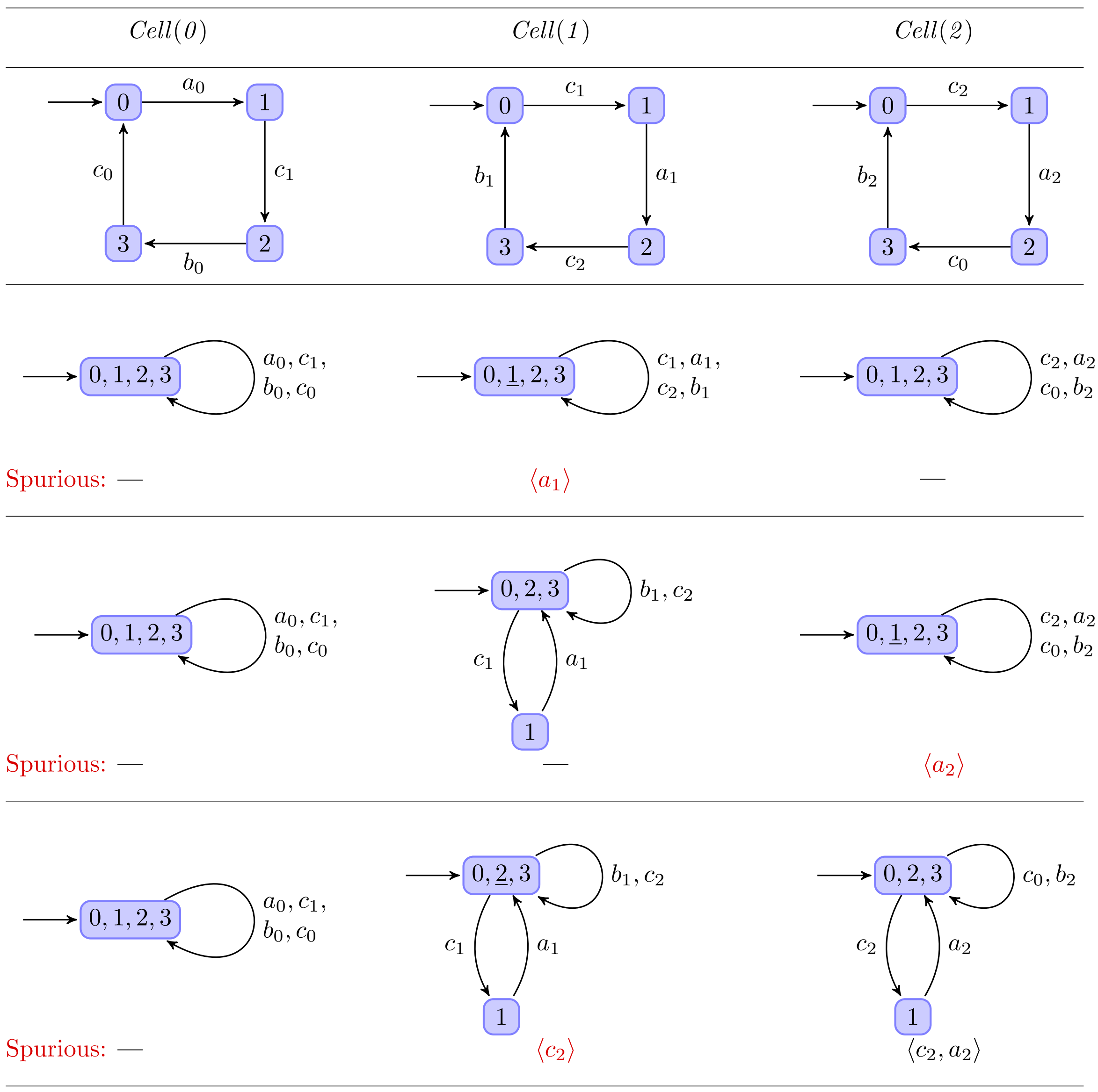


Figure 5.7: CEGAR loop for Milner's scheduler: part I

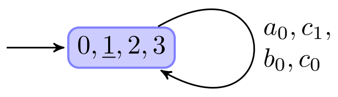
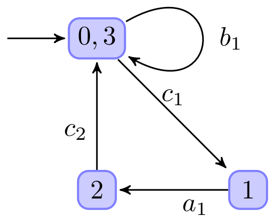
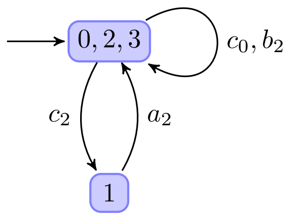
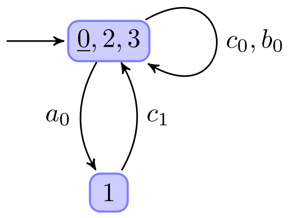
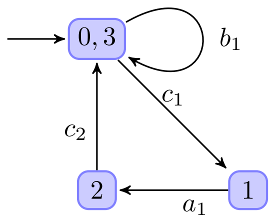
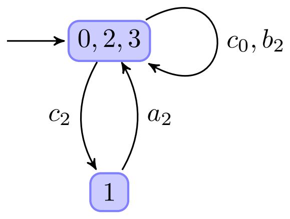
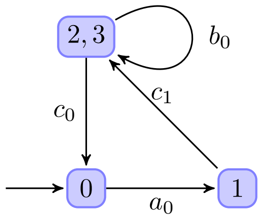
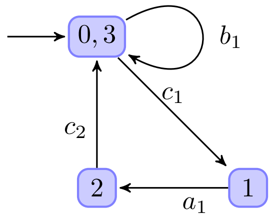
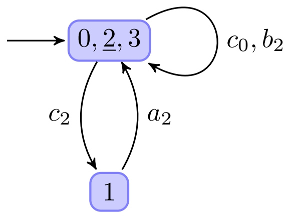
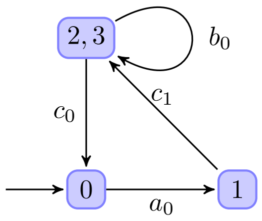
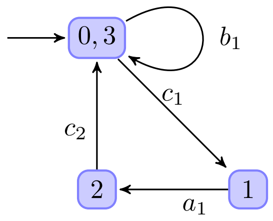
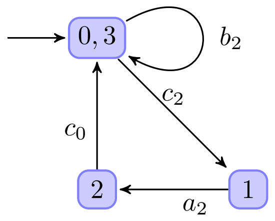
Cell(0)	*Cell(1)*	*Cell(2)*
 Spurious: $\langle c_1 \rangle$	 $\langle c_1, a_1 \rangle$	 —
 Spurious: $\langle a_0, c_1, a_0 \rangle$	 $\langle c_1 \rangle$	 —
 Spurious: $\langle a_0, c_1, c_0, a_0 \rangle$	 $\langle c_1 \rangle$	 $\langle c_0 \rangle$
 Property holds!		

Figure 5.8: CEGAR loop for Milner's scheduler: part II

Chapter 6

Conclusions and Future Work

6.1 Summary and Evaluation

The aim of this work was to develop novel verification techniques for reasoning about concurrent systems modelled in CSP, as well as to integrate those techniques in the refinement checker FDR and experimentally evaluate their efficiency and impact on combating the state-space explosion problem. We focused on compositional and/or symbolic verification techniques, which hold a lot of promise. Our directions of research broadly fell into three categories: bounded and unbounded trace refinement using SAT, symbolic compositional static analysis for establishing livelock freedom and abstraction/refinement strategies automated in a compositional CEGAR framework. For each of these categories we implemented prototype tools, which we empirically evaluated on a set of CSP benchmarks.

SAT-based trace refinement

In Chapter 3, we presented a SAT-based framework for carrying out refinement checking in the traces model of CSP, which is sufficient for verifying safety properties. We adapted bounded model checking (BMC), that can be used for bug detection, and temporal k-induction, which builds upon BMC, aims at establishing inductiveness of properties and is capable of both bug finding and establishing the correctness of systems.

Originally, the translation of BMC to SAT was modelled as reachability of error states. In the setting of CSP refinement, we used the SAT solver to decide a harder problem—bounded language inclusion. The latter is generally in PSPACE and is readily transformable to an input for a QBF solver but not for a SAT solver due to the presence of implicit universal quantifiers. To overcome this limitation, exacerbated by the presence of invisible τ actions, we reduced the problem to the bounded reachability setting by using watchdog transformations. Essentially, this involved reducing a refinement check to analysing a single process constructed by putting the implementation process in parallel with a transformed specification process. The latter played the role of a watchdog that monitored and marked violating behaviours, also flagging error states.

In Section 3.4, we described a new Boolean encoding of CSP processes based on FDR's hybrid two-level approach for operational representation based on supercombinators. We demonstrated how to glue together encodings of sequential components into an encoding of a composite concurrent system, while avoiding multiple levels of nesting, as in fully compositional encodings.

In Section 3.5, we introduced our tool SymFDR that features both bounded trace refinement and temporal k-induction and builds upon FDR to obtain an alternative symbolic refinement engine, with configurable support for an incremental SAT solver (MiniSAT, PicoSAT or ZChaff), Boolean encoding (one-hot or binary), traversal mode (forward, backward or simultaneous forward/backward), k-induction algorithm ("Zig-Zag" or "Dual").

A crucial feature of SymFDR was that in BMC mode it was also possible to configure the SAT invocation frequency, which specified how often a SAT check was run to look for errors, relative to the number of steps of unfolding the transition relation. In the original version of BMC, the system was unwound step by step until the predefined bound was reached, i.e., the SAT frequency used was 1. Despite the tremendous advances in SAT-solvers' incremental and learning capabilities, we observed that the bottleneck of the bounded refinement procedure was the SAT solver. Hence, SymFDR could jump multiple steps at once before checking whether any of those steps could lead to an error, provided that the system never deadlocked on an erroneous trace, which we guaranteed by the watchdog construction.

In Section 3.6, we demonstrated the feasibility of integrating SAT-based BMC and k-induction in FDR, and more specifically, of exchanging the expensive explicit state-space traversal phase in FDR by a SAT check in SymFDR. To the best of our knowledge, our evaluation of the application of temporal k-induction to concurrent systems was the first one to appear in literature.

In BMC mode, when finding counterexamples, SymFDR sometimes significantly outperformed FDR and even coped with problems that were beyond FDR's capabilities. In our experience, those were generally large complex tightly-coupled combinatorial problems for which a solution existed, the longest solution was relatively short (up to approximately 50–60) and was predictable in advance. In those cases, we could fix the SAT frequency close to a sizeable divisor of this length and thus spare large SAT overhead. The search space of those problems could be characterised as very wide (with respect to BFS), but relatively shallow, i.e., with a relatively small forward radius, consequence from the tight coupling.

In k-induction mode, when trying to prove correctness, the completeness threshold (the forward or backward recurrence radius) blew up in all cases, due to concurrency, and, therefore, high performance depended on whether or not the property was k-inductive for some small value of k. Hence, the explicit tool FDR performed orders of magnitude better. Nevertheless, we observed that applying compression techniques on atomic component processes was often beneficial for reducing the recurrence radius, resulting in faster convergence and better performance. Also, our experiments indicated that the backward algorithm, aiming to reach the forward recurrence radius, often scaled better than the forward one [POR12].

Our conclusions were that SAT technology was worthwhile incorporating in FDR for the sake of detecting bugs—the SAT techniques were beneficial mostly in cases when counterexamples existed. Our first attempt of efficiently establishing correctness of systems using SAT did not produce the desired results because, due to concurrency, the completeness threshold often blew up, which resulted in intractable SAT instances. The reason for this is that, similarly to the state-space explosion problem, every subsequent parallel component potentially adds a further exponential blow up to the recurrence radii of the system. In Section 6.2 we describe other SAT-based model checking techniques that we plan to incorporate in order to boost the performance of unbounded SAT-based refinement checking.

Static analysis framework for establishing livelock freedom

In Chapter 4, we introduced a novel static analysis framework for establishing livelock freedom of CSP processes. Our framework employed a collection of rules on the syntactic structure of a process to either soundly classify a process as livelock free or report an inconclusive result, thereby trading accuracy for speed. We proposed a general framework, based on reasoning about fixed points in terms of metric spaces, and a framework for finite-state processes, predicated upon analysis of fairness and co-fairness.

In the standard metric on processes, the hiding operator fails to be nonexpansive. In Section 4.5, we defined a new family of metrics parametrised by sets of visible events, under which all CSP operators other than recursion were at least nonexpansive in all their arguments, including hiding. We proved that our semantic model equipped with our new metric formed a complete ultrametric space, the set of livelock-free processes being a closed subset thereof.

In Section 4.6, we presented our general framework that was able to handle the widest variety of CSP process, including infinite-state ones. We introduced a system of rules based on the syntactic structure of CSP terms which inductively generated a sound approximation of the sets of metrics that witnessed the existence of unique and livelock-free fixed points.

In Section 4.7, we defined a class of structurally finite-state processes, for which we introduced a simpler, more efficient and more precise algorithm, capable of identifying a strictly larger class of processes as livelock free. We proposed a system of compositional rules for inductively generating a livelock flag together with a fair/co-fair characterisation of the infinite traces of a process. The algorithm benefited from being able to identify the minimal closed sequential components and examine their transition systems in isolation. For those it computed exact data and started becoming conservative only in the compositional rules for handling compound CSP processes, thereby allowing more elaborate and finer data to be computed efficiently. We proposed an overall algorithm that employed the more precise framework upon establishing that the input process was structurally finite-state, and the general framework otherwise.

In Section 4.8, we introduced our fully symbolic static analyser SLAP with support for using a SAT engine (based on MiniSAT), a BDD engine (based on CUDD), or running a SAT and a BDD analyser in parallel and reporting the results of the first one to finish. We highlighted the significance of our symbolic approach for handling efficiently potentially huge bottlenecks in our algorithms. Given a transition system of a process, we suggested an algorithm for computing the set of fair/co-fair pairs of events of the process and proposed methods for efficiently encoding this algorithm into a symbolic circuit of size polynomial in the syntactic description of the process. We also demonstrated how the entire static analysis could be carried out symbolically by providing a BDD and a SAT encoding of the collection of rules.

We note that the symbolic circuit construction that we presented in Section 4.8 is not bound to our specific process-algebraic framework and can be generalised and applied to various other contexts. A compact symbolic circuit can encode the input-output relationship of a PTIME algorithm for all possible inputs of the algorithm all at once. A translation of such a circuit into a BDD or into an input for SAT can be plugged into any symbolic implementation. We consider the approach highly beneficial and one of the key factors for the high performance of SLAP.

In Section 4.9, we illustrated the efficiency of SLAP by comparing it to the performance of FDR. In all test cases, internal communications were hidden so that livelock freedom could be viewed as progress or liveness property. The experiments indicated that our symbolic conservative approach was substantially more efficient than exhaustive search—SLAP outperformed FDR by multiple orders of magnitude, exhibiting a low rate of inconclusive results on a wide range of benchmarks. In terms of precision, SLAP was successfully able to establish livelock freedom of several network communication protocols, e.g., the alternating bit protocol and the sliding window protocol, where synchronisation of parallel components plays a subtle but vital role for ruling livelock out. In addition, due to the intricate reasoning based on metric spaces, SLAP was able to also efficiently handle infinite-state processes, which are beyond the grasp of FDR.

We concluded that when modelling systems in practice, it makes sense to try to check for livelock freedom using a simple and highly-economical static analysis before invoking computationally-expensive state-space exploration algorithms. Hence, we integrated the framework for analysing structurally finite-state processes in FDR, where it now constitutes an alternative back-end for establishing livelock freedom.

We also note that although our analysis is focused on CSP, it could be readily applied to other process algebras and, perhaps, other concurrent formalisms, in general.

Abstraction schemes and CEGAR framework

In Chapter 5, we developed a series of abstraction/refinement schemes for the traces, stable-failures and failures-divergences model of CSP, and demonstrated how to embed them into a fully automated and compositional counterexample-guided abstraction refinement framework (CEGAR).

In Section 5.2, we sketched the general skeleton of our CEGAR loop in which successive abstraction refinements converge, in the worst case, to strong or DRW bisimulation quotients of the systems under consideration. We proposed strategies for carrying out the stages of initial abstraction, counterexample validation and abstraction refinement component-wisely on leaf-process level, facilitated by the capabilities of FDR. This was justified by theoretical results for the monotonicity and transitivity of CSP operators and the fact that supercombinators act on a higher level to control how leaf processes interact.

In Section 5.3, we presented a series of abstraction schemes for the traces, stable-failures and failures-divergences model of CSP. Our abstractions were identified by partitions of the state spaces of the leaf processes, coupled with minimal existential abstraction and annotations for preserving nondeterminism-related information. Since existential abstraction was inadequate for preserving liveness properties, we extended the proposal in [CCOS04, CCO$^+$05] to define abstract minimal acceptances and abstract divergences. We observed that abstracting models based on traces and abstracting models based on stable failures were triggered by two opposite forces and proposed abstraction strategies for balancing those forces.

For establishing whether or not an abstract counterexample behaviour is spurious, model checkers generally employ explicit or implicit simulation techniques or theorem provers. In Section 5.4, we proposed an approach for efficiently performing counterexample validation by devising suitable refinement checks in FDR. We suggested methods for validating finite and lasso traces, divergence traces and stable failures, for all three semantic models that we considered.

In Section 5.5, we presented strategies for obtaining all abstract executions of a given spurious behaviour by employing refinement checks in FDR. We also introduced and a number of abstraction refinement schemes for all types of counterexample behaviours—finite and lasso traces, divergence traces and stable failures.

In Section 5.7, we reported on some preliminary experimental results with a prototype tool developed on top of FDR. The tool offers configurable support for the abstraction and refinement strategies, the number of leaf processes to refine on each iteration of the CEGAR loop (just one or all spurious), etc. Generally, we adopted lazy refinement strategies that yielded coarser abstractions even though it took a greater number of iterations to converge. Experiments indicated a significant enhancement in terms of performance when verifying both safety and liveness properties, including checks for livelock and deadlock. As a major limitation, we observed that different test cases benefited/deteriorated from different abstraction/refinement strategies and number of leaves to refine. Hence, we speculate that randomising the choice of those would produce consistently more favourable results.

6.2 Future Work

We envision multiple directions for enhancing and augmenting the work presented in this thesis. In addition to extending the three frameworks that we presented, we plan to investigate methods for applying a number of the underlying techniques in conjunction in order to combine and amplify their advantages. We also consider incorporating a number of other compositional model-checking techniques for tackling the challenges arising from concurrency.

For the SAT refinement framework, we intend to focus on applying alternative techniques for obtaining a complete SAT-based refinement checker. As stated in Section 3.7, we plan to incorporate Craig-interpolation techniques [McM03, McM06] and proof-obligation methods [EMB11, Bra11], both of which have proven more successful in practice. Both methodologies have been little evaluated in the context of concurrent systems, even less so for message-passing models of concurrency. Therefore we would be able to establish whether or not those techniques are scalable and beneficial mainly after implementing and empirically evaluating their performance. For the Craig interpolation framework, we also plan to experiment with applying interpolants of different quality and strength as described in [DKPW10] and with incorporating techniques for invariant strengthening [BHvMW09] in order to decrease the completeness threshold values. The latter can also be used for improving the efficiency of the temporal k-induction algorithm.

For moderating the severity of the state-space explosion problem, we plan to explore in more depth partial-order reduction methods targeted to concurrent systems. Approaches exploiting the interleaving model of concurrent systems and the notion of independence of concurrent events include the ample sets of Peled [Pel98], the persistent sets of Godefroid [God95] and the stubborn sets of Valmari [Val97]. We also plan to investigate methods for efficiently coupling partial-order reductions with SAT and a good starting point towards this goal would be the work presented in [KWG09, Kah12], which however has been based on shared-variable concurrency models and state-based formalisms.

Another approach combining iterative approximation and SAT techniques would be to implement the proof-guided underapproximation widening algorithm [GLST05], which is the mirror-opposite dual version of CEGAR and can be developed on top of a BMC implementation. In this framework, the system is checked against a series of increasingly less constrained underapproximations that gradually amplify the amount of concurrency allowed in the system. Counterexamples detected during the model checking phase are guaranteed to be genuine due to underapproximation. However, establishing correctness might be spurious, i.e., a "false positive", the check of which can be carried out automatically using the proof of unsatisfiability generated by the SAT solver for the positive BMC instance. In this case the system needs to be widened by adding more behaviours which eliminate the "false positive" correctness result.

For all SAT-related tools, we would be looking forward to further progress in the area of SAT technology. So far, only MiniSAT 2.0 has been able to tractably analyse encodings of complex concurrent CSP processes. All the other state-of-the-art SAT-solvers that we experimented with (PicoSAT, ZChaff and MiniSAT 1.14) were substantially less efficient. Unfortunately, MiniSAT 2.0, as well as newer versions, do not yet provide utilities for extracting an unsatisfiable core of clauses and a proof of unsatisfiability, which are prerequisites for implementing the Craig-interpolation framework and the proof-guided underapproximation widening algorithm. The main difficulties stem from the usage of variable and clause elimination in the preprocessing phase [EB05] which modifies the unsatisfiable core and the proof of unsatisfiability. Without this optimisation, though, SymFDR with MiniSAT 2 does not scale well either.

The CEGAR framework for CSP and FDR can also be extended in multiple ways. We plan to investigate strategies for abstraction refinement that further minimize the blow-up of minimal acceptances of abstract states. This would involve identifying structural refinements between subsets of concrete states. Currently we have a plethora of options for abstraction refinement for each possible type of behaviour. However, for a given test case, we have little knowledge and experience in predicting which strategy would be beneficial and which would worsen the performance of the tool. To combat this, we plan to develop heuristics for choosing a suitable abstraction refinement strategy or, alternatively, use the power of randomization to do so.

As more general extensions of the CEGAR framework, we plan to further incorporate different abstraction mechanisms, e.g., data abstraction, counter abstraction, etc. Furthermore, different classes of simulation relations can be exploited to obtain equivalence relations which are weaker than bisimulation but stronger than trace equivalence [vG00, BK08]. Algorithms for simulation quotienting [BK08] can then be easily embedded in the CEGAR framework that we presented and we expect greater state-space reduction compared to their bisimulation quotienting counterparts. An exciting line of research would be to also combine CEGAR with assume-guarantee reasoning along the lines of [CCGP, BPG08].

The static analysis framework for establishing livelock freedom sometimes over-conservatively marks certain CSP processes as potentially divergent even though they are livelock free. Some directions for future work include improving the accuracy of our analysis without compromising the efficiency and vice versa. In terms of accuracy, we plan to experiment with applying algebraic laws at the syntactic level, such as bounded unfoldings of parallel compositions. To boost efficiency, options include incorporating certain abstraction mechanisms, e.g., collapsing all events on a given channel, or placing bounds on the size of sets that the static analyser generates.

As non-immediate extensions of the static analyser, we have ideas how to formulate similar systems of rules for establishing deadlock freedom as well. Moreover, we would like to study more carefully the links between livelock freedom and general termination analysis [CY10]. As a more ambitious goal, we would like to investigate more closely how we can transfer our analyses to other concurrent formalisms such as other process algebras or maybe even shared-variable frameworks.

Appendix A

Proofs

A.1 Proofs for Section 4.4.2.2

Throughout the section we will use the following notation. For every $u \in \Sigma^\omega$ and $i \in \mathbb{N}$ we will denote by u_i the prefix of u of length i. Then, as explained in Section 4.4.2.2, $u \in \mathsf{traces}^\omega(P)$ if and only if for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P) \cap \Sigma^*$. Let us recall that for every $i \in \mathbb{N}$, u_i cannot contain a $\checkmark$ and is therefore an element of Σ^*. We will frequently make use of the following observation which relies on the set $\mathsf{traces}(P)$ being prefix-closed. If $u_i \in \mathsf{traces}(P)$ for infinitely many $i \in \mathbb{N}$, then $u_i \in \mathsf{traces}(P)$ for all $i \in \mathbb{N}$, and therefore $u \in \mathsf{traces}^\omega(P)$. Most proofs will be based on König's Lemma, which we now recall.

Theorem A.1.1 (König's Lemma). *Suppose that for each $i \in \mathbb{N}$, X_i is a non-empty finite set and $f_i : X_{i+1} \to X_i$ is a total function. Then there is a sequence $\langle x_i \mid i \in \mathbb{N}\rangle$, such that $x_i \in X_i$ and $f_i(x_{i+1}) = x_i$.* $\square$

In our proofs we will define the sets X_i as specific subsets of $\mathsf{traces}(P) \cap \Sigma^i$. For each $i \in \mathbb{N}$, $x_i \in X_i$ and $x_{i+1} \in X_{i+1}$, $f_i(x_{i+1}) = x_i$ will imply that $x_i < x_{i+1}$, where $<$ denotes the strict prefix order on Σ^*. For a given $x_{i+1} \in X_{i+1}$, the choice for $f_i(x_{i+1})$ might not be unique, but we can take an arbitrary prefix x_i of x_{i+1} from X_i satisfying certain properties. Then the sequence $\langle x_i \mid i \in \mathbb{N}\rangle$ will form an infinite chain $x_0 < x_1 < x_2 < \dots x_n \dots$ under prefix and $x = \lim_{i=0}^{\infty} x_i \in \mathsf{traces}^\omega(P)$.

Lemma 4.4.2. *Let $u \in \mathsf{traces}^\omega(a \longrightarrow P)$. Then there exists $u' \in \mathsf{traces}^\omega(P)$, such that $u = \langle a\rangle^\frown u'$.*

Proof. Let $u \in \mathsf{traces}^\omega(a \longrightarrow P)$. By definition, for each $t < u$, $t \in \mathsf{traces}(a \longrightarrow P)$. Then, for each $t < u$, $t = \langle a \rangle^\frown t'$ for some $t' \in \mathsf{traces}(P)$. Let $u = \langle a \rangle^\frown u'$ for some $u' \in \Sigma^\omega$. Then, for each $t' < u'$, $t' \in \mathsf{traces}(P)$. Therefore, by definition, $u' \in \mathsf{traces}^\omega(P)$. □

Lemma 4.4.3. *Let $u \in \mathsf{traces}^\omega(P \oplus Q)$ for $\oplus \in \{\Box, \sqcap\}$. Then $u \in \mathsf{traces}^\omega(P)$ or $u \in \mathsf{traces}^\omega(Q)$.*

Proof. Let $u \in \mathsf{traces}^\omega(P \oplus Q)$. By definition, for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P \oplus Q)$. Therefore, for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P)$ or $u_i \in \mathsf{traces}(Q)$. Then, due to the pigeonhole principle, $u_i \in \mathsf{traces}(P)$ for infinitely many $i \in \mathbb{N}$ or $u_i \in \mathsf{traces}(Q)$ for infinitely many $i \in \mathbb{N}$. Let without loss of generality the former holds. Then, $u_i \in \mathsf{traces}(P)$ for all $i \in \mathbb{N}$, and hence $u \in \mathsf{traces}^\omega(P)$. □

Lemma 4.4.4. *Let $u \in \mathsf{traces}^\omega(P \mathbin{;} Q)$. Then $u \in \mathsf{traces}^\omega(P)$, or $u = t^\frown u'$ with $t^\frown\langle \checkmark \rangle \in \mathsf{traces}(P)$ and $u' \in \mathsf{traces}^\omega(Q)$.*

Proof. Let $u \in \mathsf{traces}^\omega(P \mathbin{;} Q)$. By definition, for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P \mathbin{;} Q)$. Therefore, for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P)$ or $u_i = t_1{}^\frown t_2$ with $t_1{}^\frown\langle \checkmark \rangle \in \mathsf{traces}(P) \cap \Sigma^{*\checkmark}$, $t_2 \in \mathsf{traces}(Q) \cap \Sigma^*$. If for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P)$, then, by definition, $u \in \mathsf{traces}^\omega(P)$. Otherwise, there exists $N \in \mathbb{N}$, such that $u_0, u_1, \ldots, u_N \in \mathsf{traces}(P)$, but $u_{N+1} \notin \mathsf{traces}(P)$. Therefore, for $j \geq 1$, $u_{N+j} \notin \mathsf{traces}(P)$. By assumption, for every $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P \mathbin{;} Q)$. Therefore, for $j \geq 1$, $u_{N+j} = t_j{}^\frown v_j$ where $t_j{}^\frown\langle \checkmark \rangle \in \mathsf{traces}(P)$ (and therefore $t_j \leq u_N$) and $v_j \in \mathsf{traces}(Q)$. Then, there must be some $t \leq u_N$, such that $t_j = t$ for infinitely many u_{N+j}'s. Let us write $u_j = t^\frown w_j$ for $j \geq |t|$. We have that $t^\frown\langle \checkmark \rangle \in \mathsf{traces}(P)$ and infinitely often $w_j \in \mathsf{traces}(Q)$. Since for $j < j'$, $w_j < w_{j'}$, and the set of traces is prefix-closed, $w_j \in \mathsf{traces}(Q)$ for each $j \geq |t|$. Then, by definition, $u' = \lim_{j=|t|}^{\infty} w_j \in \mathsf{traces}^\omega(Q)$. □

Lemma 4.4.5. *Let $u \in \mathsf{traces}^\omega(P \setminus A)$ and $P \setminus A$ be livelock-free. Then there exists $v \in \mathsf{traces}^\omega(P)$, such that $u = v \upharpoonright (\Sigma \backslash A)$.*

Proof. Let $u \in \mathsf{traces}^\omega(P \setminus A)$. By definition, for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P \setminus A)$, i.e., there exists $v_{j_i} \in \mathsf{traces}(P)$, such that $u_i = v_{j_i} \upharpoonright (\Sigma \backslash A)$.

Let, for $i \in \mathbb{N}$, $\backslash^{-1}(u_i) = \{v \in \mathsf{traces}^{\infty}(P) \mid v \upharpoonright (\Sigma \backslash A) = u_i\}$. We claim that, for each $i \in \mathbb{N}$, $\backslash^{-1}(u_i)$ is finite. Suppose, for the sake of the argument, that there exists $k \in \mathbb{N}$ such that $\backslash^{-1}(u_k)$ is infinite. We will prove that $P \setminus A$ is divergent, which will be a contradiction with $P \setminus A$ being livelock-free. Let $u_k = \langle a_1, a_2, \ldots, a_k \rangle$. It is clear that $\{a_1, a_2, \ldots, a_k\} \cap A = \emptyset$. Then, $\backslash^{-1}(u_k) = ((A^* \cup A^{\omega})\ a_1\ (A^* \cup A^{\omega})\ a_2\ (A^* \cup A^{\omega}) \ldots (A^* \cup A^{\omega})\ a_k\ (A^* \cup A^{\omega})) \cap \mathsf{traces}^{\infty}(P)$. Let for $i \in \{0, \ldots, k-1\}$, n_i be the maximum number of occurrences of consecutive events from A before the occurrence of a_{i+1}, and let n_k be the maximum number of consecutive events from A after a_k. Then, for $i = \{0, \ldots, k\}$, $n_i \in \mathbb{N} \cup \{\omega\}$. Since $\backslash^{-1}(u_k)$ is infinite, there exists $j \in \{0, \ldots, k\}$, such that $n_j = \omega$. Let $j_{\min}$ be the minimal j with this property. Then, for $i < j_{\min}$, $n_i \in \mathbb{N}$. Let $v \in (A^* a_1 A^* a_2 A^* \ldots A^* a_{j_{\min}} A^{\omega}) \cap \mathsf{traces}^{\infty}(P)$. Therefore, $v \upharpoonright (\Sigma \backslash A) = \langle a_1, a_2, \ldots a_{j_{\min}} \rangle = u_{j_{\min}} \in \mathsf{divergences}(P \setminus A)$, which is a contradiction with $P \setminus A$ being livelock-free. Hence, for $i = \{0, \ldots, k\}$, $n_i \in \mathbb{N}$, and therefore $\backslash^{-1}(u_k)$ is finite. Therefore, for each $i \in \mathbb{N}$, we have:

1. $\backslash^{-1}(u_i) \neq \emptyset$ because $u_i \in \mathsf{traces}(P \setminus A)$.

2. $\backslash^{-1}(u_i)$ is finite.

3. For each $j > i$, for each $w \in \backslash^{-1}(u_j)$, there exists $v \in \backslash^{-1}(u_i)$, such that $v < w$. The trace v can be defined as an arbitrary prefix of w of $(\Sigma \backslash A)$-length i.

Therefore, by König's Lemma, there exists an infinite sequence $v_{j_1} < v_{j_2} < \ldots < v_{j_n} < \ldots$, such that for $i \in \mathbb{N}$, $v_{j_i} \in \backslash^{-1}(u_i)$, i.e., $v_{j_i} \in \mathsf{traces}(P)$ and $u_i = v_{j_i} \upharpoonright (\Sigma \backslash A)$. Therefore, $v = \lim_{i=0}^{\infty} v_{j_i} \in \mathsf{traces}^{\omega}(P)$ and $u = v \upharpoonright (\Sigma \backslash A)$. □

Lemma 4.4.6. *Let $u \in \mathsf{traces}^{\omega}(P[\![R]\!])$. Then there exists $v \in \mathsf{traces}^{\omega}(P)$, such that $v\ R\ u$.*

Proof. Let $u \in \mathsf{traces}^{\omega}(P[\![R]\!])$. By definition, for each $i \in \mathbb{N}$, $u_i \in \mathsf{traces}(P[\![R]\!]) \cap \Sigma^*$. Therefore, for each $i \in \mathbb{N}$, there exists $v_{j_i} \in \mathsf{traces}(P) \cap \Sigma^*$, such that $v_{j_i}\ R\ u_i$, i.e., $\mathsf{length}(u_i) = \mathsf{length}(v_{j_i}) = i$ and for each $0 \leq k \leq i$, $v_{j_i}(k)\ R\ u_i(k)$. Let, for $i \in \mathbb{N}$, $R^{-1}(u_i) = \{v \in \mathsf{traces}(P) \mid v\ R\ u_i\}$. Then, for $i \in \mathbb{N}$:

1. $R^{-1}(u_i) \neq \emptyset$ because $u_i \in \mathsf{traces}(P[\![R]\!])$.

2. $R^{-1}(u_i)$ is finite because Σ, and therefore R, are finite.

3. For each $j > i$ and each $w \in R^{-1}(u_j)$, there exists $v \in R^{-1}(u_i)$, such that $v < w$. The trace v can be constructed as the prefix of w of length i.

Therefore, by König's Lemma, there exists an infinite sequence $v_{j_1} < v_{j_2} < \ldots < v_{j_n} < \ldots$, such that for $i \in \mathbb{N}$, $v_{j_i} \in R^{-1}(u_i)$, i.e., $v_{j_i} \in \mathsf{traces}(P)$ and $v_{j_i}\ R\ u_i$. Therefore, $v = \lim_{i=0}^{\infty} v_{j_i} \in \mathsf{traces}^{\omega}(P)$ and $v\ R\ u$. □

Lemma 4.4.7. *Let* $u \in \mathsf{traces}^{\omega}(P \underset{A}{\parallel} Q)$*. Then there exist* $u_1 \in \mathsf{traces}^{\infty}(P)$*,* $u_2 \in \mathsf{traces}^{\infty}(Q)$*, such that* $u \in u_1 \underset{A}{\parallel} u_2$*, and* $u_1 \in \Sigma^{\omega}$ *or* $u_2 \in \Sigma^{\omega}$*.*

Proof. Let $u \in \mathsf{traces}^{\omega}(P \underset{A}{\parallel} Q)$. Then, for each $n \in \mathbb{N}$, $u_n \in \mathsf{traces}(P \underset{A}{\parallel} Q) \cap \Sigma^*$. Therefore, by definition, for each $n \in \mathbb{N}$, there exist $v_{i_n} \in \mathsf{traces}(P) \cap \Sigma^*$ and $w_{j_n} \in \mathsf{traces}(Q) \cap \Sigma^*$, such that $u_n \in v_{i_n} \underset{A}{\parallel} w_{j_n}$ and $n \leq |v_{i_n}| + |w_{j_n}| \leq 2n$. Therefore, for each such triple (u_n, v_{i_n}, w_{j_n}) there exists a function $f^n : \{1, \ldots, n\} \mapsto \{0, 1, 2\}$ specifying a possible interleaving of v_{i_n} and w_{j_n} for obtaining u_n. More specifically, $f^n(i)$ indicates which process contributes for communicating the i-th event of u_n, with 0 denoting both P and Q (for events in A), 1 denoting only P, and 2 denoting only Q. Given $u_n = \langle a_1, \ldots a_n \rangle$ and f^n, v_{i_n} and w_{j_n} are identified uniquely as $v_{i_n} = \langle a_i \mid 1 \leq i \leq n, f^n(i) \subseteq \{0, 1\} \rangle$, $w_{j_n} = \langle a_j \mid 1 \leq j \leq n, f^n(j) \subseteq \{0, 2\} \rangle$.

Let us define a partially-ordered set $((\Sigma^{*\checkmark})^2, \leq)$ with $(v, w) \leq (v', w')$ iff $v \leq v'$ and $w \leq w'$, where $\leq$ denotes a non-strict prefix on traces. We will prove that there exists an infinite chain $(v_{i_1}, w_{j_1}) \leq \ldots \leq (v_{i_n}, w_{j_n}) \leq \ldots$, such that for each $n \in \mathbb{N}$, $v_{i_n} \in \mathsf{traces}(P)$, $w_{j_n} \in \mathsf{traces}(Q)$ and $u_n \in v_{i_n} \underset{A}{\parallel} w_{j_n}$.

Let for $k \in \mathbb{N}$, $\underset{A}{\parallel}^{-1}(u_k) = \{(v_{i_k}, w_{j_k}) \mid v_{i_k} \in \mathsf{traces}(P), w_{j_k} \in \mathsf{traces}(Q), u_k \in v_{i_k} \underset{A}{\parallel} w_{j_k}\}$. Then:

1. $\underset{A}{\parallel}^{-1}(u_k) \neq \emptyset$ because $u_k \in \mathsf{traces}(P \underset{A}{\parallel} Q)$.

2. $\underset{A}{\parallel}^{-1}(u_k)$ is finite because Σ is finite.

3. For each $l < k$ and each $(v_{i_k}, w_{j_k}) \in \underset{A}{\parallel}^{-1}(u_k)$, there exists $(v_{i_l}, w_{j_l}) \in \underset{A}{\parallel}^{-1}(u_l)$, such that $(v_{i_l}, w_{j_l}) < (v_{i_k}, w_{j_k})$. The pair of traces (v_{i_l}, w_{j_l}) can be constructed as follows.

Let for the triple (u_k, v_{i_k}, w_{j_k}) the function $f^k : \{1, \ldots, n\} \mapsto \{0, 1, 2\}$ specifies a possible interleaving of v_{i_k} and w_{j_k} for obtaining u_k. We define $f^l(i) = f^k(i)$ for $1 \leq i \leq l$. Then, (v_{i_l}, w_{j_l}) is the pair that is uniquely identified by f^l and u_l.

Therefore, by König's Lemma, for each $n \in \mathbb{N}$, there exists a chain $(v_{i_1}, w_{j_1}) \leq (v_{i_2}, w_{j_2}) \leq \ldots \leq (v_{i_n}, w_{j_n})$, such that for each $1 \leq k \leq n$, $v_{i_k} \in \mathsf{traces}(P)$, $w_{j_k} \in \mathsf{traces}(Q)$, $u_k \in v_{i_k} \underset{A}{\parallel} w_{j_k}$ and $k \leq |v_{i_k}| + |w_{j_k}| \leq 2k$. Let $v = \lim_{k=1}^{\infty} v_{i_k}$ and $w = \lim_{k=1}^{\infty} w_{j_k}$. Then clearly, $v \in \mathsf{traces}^{\infty}(P)$, $w \in \mathsf{traces}^{\infty}(Q)$ and $u \in v \underset{A}{\parallel} w$. Let us assume that both v and w are finite, i.e., $|v| = l_v$ and $|w| = l_w$, for some $l_v, l_w \in \mathbb{N}$. Then, each prefix of u will be of length at most $l_v + l_w \in \mathbb{N}$, which is a contradiction with u being infinite. Therefore, at least one of v and w is infinite. $\square$

A.2 Proofs for Section 4.5.2

Lemma A.2.1 ([Sut75, Lemma 9.2.5])**.** *In any metric space, if s is a Cauchy sequence that has a subsequence that converges to a point x, then s also converges to x.*

Proposition A.2.2. *Let $U \subseteq \Sigma$. Then $\mathcal{T}^{\Downarrow}$ equipped with the metric d_U is a complete metric space.*

Proof. We will prove that every Cauchy sequence converges.

Let $\langle P_i \mid i \in \mathbb{N}\rangle$ be a Cauchy sequence in $(\mathcal{T}^{\Downarrow}, d_U)$. By definition, for every $\varepsilon > 0$, there exists $N_\varepsilon \in \mathbb{N}$ such that, for every $n, m \geq N_\varepsilon$, $d_U(P_n, P_m) < \varepsilon$. Therefore, for every $r \in \mathbb{N}$ and $\varepsilon = 2^{-r}$, there exists $N_r \in \mathbb{N}$ such that, for every $n, m \geq N_r$, $d_U(P_n, P_m) < 2^{-r}$, i.e., $P_n \upharpoonright_U r = P_m \upharpoonright_U r$. Then, for every $r, m \in \mathbb{N}$, $d_U(P_{N_r}, P_{N_{r+m}}) < 2^{-r}$. Therefore, the subsequence $\langle P_{N_r} \mid r \in \mathbb{N}\rangle$ of $\langle P_i \mid i \in \mathbb{N}\rangle$ is itself a Cauchy sequence.

Let us define $P = \bigsqcap_{q\in\mathbb{N}} \bigsqcup_{r\geq q} P_{N_r}$. $P \in \mathcal{T}^{\Downarrow}$ because $(\mathcal{T}^{\Downarrow}, \sqsubseteq)$ is a complete lattice. We will prove that the subsequence $\langle P_{N_r} \mid r \in \mathbb{N}\rangle$ converges to P, i.e., that for every $r \in \mathbb{N}$, $d_U(P_{N_r}, P) < 2^{-r}$.

Let us fix r. Suppose, for the sake of the argument, that $d_U(P_{N_r}, P) \geq 2^{-r}$ and let, without loss of generality, P_{N_r} and P disagree on the sets of their divergences. Therefore, there exists $t \in \Sigma^{*\checkmark}$ such that $\mathsf{length}_U(t) < r$ and, either $t \in \mathsf{divergences}(P_{N_r}) \setminus \mathsf{divergences}(P)$ or $t \in \mathsf{divergences}(P) \setminus \mathsf{divergences}(P_{N_r})$. To recall, by construction we have $\mathsf{divergences}(P) = \bigcup_{q\in\mathbb{N}} \bigcap_{r\geq q} \mathsf{divergences}(P_{N_r})$. We explore both alternatives.

- Suppose $t \in \mathsf{divergences}(P_{N_r}) \setminus \mathsf{divergences}(P)$. Since $t \notin \mathsf{divergences}(P)$, for every $q \in \mathbb{N}$ there exists $s_q \geq q$ such that $t \notin \mathsf{divergences}(P_{N_{s_q}})$. Therefore, for $q = r$ there exists $s_r \geq r$ such that $t \notin \mathsf{divergences}(P_{N_{s_r}})$. Hence, since $t \in \mathsf{divergences}(P_{N_r})$ and $\mathsf{length}_U(t) < r$, we obtain $d_U(P_{N_r}, P_{N_{s_r}}) \geq 2^{-r}$, which is a contradiction with $d_U(P_{N_r}, P_{N_{r+m}}) < 2^{-r}$ for $m \geq 0$.

- Suppose $t \in \mathsf{divergences}(P) \setminus \mathsf{divergences}(P_{N_r})$. Since $t \in \mathsf{divergences}(P)$, there exists $q \in \mathbb{N}$ such that for every $s \geq q$, $t \in \mathsf{divergences}(P_{N_s})$. However, as $t \notin \mathsf{divergences}(P_{N_r})$ and $\mathsf{length}_U(t) < r$, for every $s \geq r$, $t \notin \mathsf{divergences}(P_{N_s})$, which again leads to a contradiction.

Therefore, for every $r \in \mathbb{N}$, $d_U(P_{N_r}, P) < 2^{-r}$, and hence the subsequence $\langle P_{N_r} \mid r \in \mathbb{N}\rangle$ converges to P. Therefore, from Lemma A.2.1, $\langle P_i \mid i \in \mathbb{N}\rangle$ also converges to P, and hence $(\mathcal{T}^{\Downarrow}, d_U)$ is a complete metric space. □

Proposition A.2.3. *Let $U \subseteq \Sigma$. Then the set of livelock-free processes is a closed subset of $(\mathcal{T}^{\Downarrow}, d_U)$.*

Proof. Let $\langle P_i \mid i \in \mathbb{N}\rangle$ be a sequence of livelock-free elements of $\mathcal{T}^{\Downarrow}$ converging to a process $Q \in \mathcal{T}^{\Downarrow}$. Therefore, by definition, for every $\varepsilon > 0$, there exists $N \in \mathbb{N}$ such that, for every $n \geq N$, $d_U(P_n, Q) < \varepsilon$. We will prove that Q is also livelock-free.

Suppose for the sake of the argument that Q can diverge. Let $t \in \mathsf{divergences}(Q)$ and $\mathsf{length}_U(t) = k$. If we take $\varepsilon = 2^{-k}$, since $\langle P_i \mid i \in \mathbb{N}\rangle$ converges to Q, there exists $N_t \in \mathbb{N}$ such that, for every $n \geq N_t$, $d_U(P_n, Q) < 2^{-k}$, and therefore $P_n \upharpoonright_U k = Q \upharpoonright_U k$. Therefore, for every $n \geq N_t$, $t \in \mathsf{divergences}(P_n)$, which is a contradiction with $\langle P_i \mid i \in \mathbb{N}\rangle$ being all livelock-free.

Therefore, Q is livelock-free, and hence the set of livelock-free processes is closed. □

A.3 Proofs for Section 4.5.3

Throughout this section let us fix a set of events $U \subseteq \Sigma$.

Lemma 4.5.7 ($\fatsemi_1$)**.** *For any CSP processes P, P' and Q,*

$$d_U(P \fatsemi Q, P' \fatsemi Q) \leq d_U(P, P').$$

Proof. Suppose $(T_P, D_P) \restriction_U k = (T_{P'}, D_{P'}) \restriction_U k$. We will prove that $(T_{P \fatsemi Q}, D_{P \fatsemi Q}) \restriction_U k = (T_{P' \fatsemi Q}, D_{P' \fatsemi Q}) \restriction_U k$, from which we can conclude that $d_U(P \fatsemi Q, P' \fatsemi Q) \leq d_U(P, P')$.

Let $t \in \mathsf{divergences}(P \fatsemi Q)$ and $\mathsf{length}_U(t) \leq k$. We will prove that $t \in \mathsf{divergences}(P' \fatsemi Q)$, and therefore $D_{P \fatsemi Q} \restriction_U k \subseteq D_{P' \fatsemi Q} \restriction_U k$. The reverse containment is established similarly by symmetry.

Since $t \in \mathsf{divergences}(P \fatsemi Q)$, by definition, $t \in \mathsf{divergences}(P)$ or $t = t_1 {}^\frown t_2$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(P)$, $t_2 \in \mathsf{divergences}(Q)$. We consider both cases.

- Suppose $t \in \mathsf{divergences}(P)$. Since $\mathsf{length}_U(t) \leq k$ and $(T_P, D_P) \restriction_U k = (T_{P'}, D_{P'}) \restriction_U k$, $t \in \mathsf{divergences}(P')$. Therefore, by definition, $t \in \mathsf{divergences}(P' \fatsemi Q)$.

- Suppose $t = t_1 {}^\frown t_2$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(P)$, $t_2 \in \mathsf{divergences}(Q)$. Observe that $\mathsf{length}_U(t_1 {}^\frown \langle \checkmark \rangle) = \mathsf{length}_U(t_1) \leq \mathsf{length}_U(t) \leq k$. Then, since $(T_P, D_P) \restriction_U k = (T_{P'}, D_{P'}) \restriction_U k$, $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(P')$. Hence, by definition we have $t_1 {}^\frown t_2 = t \in \mathsf{divergences}(P' \fatsemi Q)$.

Now let $t \in \mathsf{traces}_\perp(P \fatsemi Q)$ and $\mathsf{length}_U(t) \leq k$. We will prove that $t \in \mathsf{traces}_\perp(P' \fatsemi Q)$ and therefore, $T_{P \fatsemi Q} \restriction_U k \subseteq T_{P' \fatsemi Q} \restriction_U k$. The reverse containment is established similarly by symmetry. Since $t \in \mathsf{traces}_\perp(P \fatsemi Q)$, $t \in \mathsf{divergences}(P \fatsemi Q)$ or $t \in \mathsf{traces}(P \fatsemi Q)$. The latter reduces to $t \in \mathsf{traces}(P) \cap \Sigma^*$, or $t = t_1 {}^\frown t_2$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(P)$, $t_2 \in \mathsf{traces}(Q)$. We consider all three alternatives.

- Suppose first that $t \in \mathsf{divergences}(P \fatsemi Q)$. We already proved that $t \in \mathsf{divergences}(P' \fatsemi Q)$ and therefore, $t \in \mathsf{traces}_\perp(P' \fatsemi Q)$.

- Suppose now that $t \in \mathsf{traces}(P) \cap \Sigma^*$. Therefore, $t \in \mathsf{traces}_\perp(P) \cap \Sigma^*$. Then, since $(T_P, D_P) \restriction_U k = (T_{P'}, D_{P'}) \restriction_U k$, $t \in \mathsf{traces}_\perp(P') \cap \Sigma^*$.

  - If $t \in \mathsf{traces}(P') \cap \Sigma^*$, then by definition, $t \in \mathsf{traces}(P' \mathbin{⨾} Q) \subseteq \mathsf{traces}_\perp(P' \mathbin{⨾} Q)$.
  - If $t \in \mathsf{divergences}(P') \cap \Sigma^*$, then by definition, $t \in \mathsf{divergences}(P' \mathbin{⨾} Q) \subseteq \mathsf{traces}_\perp(P' \mathbin{⨾} Q)$.

- Suppose finally that $t = t_1 \frown t_2$ with $t_1 \frown \langle \checkmark \rangle \in \mathsf{traces}(P)$, $t_2 \in \mathsf{traces}(Q)$. We note that $\mathsf{length}_U(t_1 \frown \langle \checkmark \rangle) = \mathsf{length}_U(t_1) \leq \mathsf{length}_U(t) \leq k$. Then, since $t_1 \frown \langle \checkmark \rangle \in \mathsf{traces}(P)$ and $(T_P, D_P) \restriction_U k = (T_{P'}, D_{P'}) \restriction_U k$, $t_1 \frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(P')$.
  - Let $t_1 \frown \langle \checkmark \rangle \in \mathsf{traces}(P')$. By definition, $t \in \mathsf{traces}(P' \mathbin{⨾} Q) \subseteq \mathsf{traces}_\perp(P' \mathbin{⨾} Q)$.
  - Let $t_1 \frown \langle \checkmark \rangle \in \mathsf{divergences}(P')$. By Axiom 2 of $\mathcal{T}^\Downarrow$, $t_1 \in \mathsf{divergences}(P')$. Since $t_1 \in \Sigma^*$, by Axiom 4 of $\mathcal{T}^\Downarrow$, $t = t_1 \frown t_2 \in \mathsf{divergences}(P')$. Then by definition, $t \in \mathsf{divergences}(P' \mathbin{⨾} Q) \subseteq \mathsf{traces}_\perp(P' \mathbin{⨾} Q)$.

Therefore, $(T_{P⨾Q}, D_{P⨾Q}) \restriction_U k = (T_{P'⨾Q}, D_{P'⨾Q}) \restriction_U k$, and hence $d_U(P \mathbin{⨾} Q, P' \mathbin{⨾} Q) \leq d_U(P, P')$. □

Lemma 4.5.7 **($⨾_2$).** *For any CSP processes P, Q and Q',*

$$d_U(P \mathbin{⨾} Q, P \mathbin{⨾} Q') \leq d_U(Q, Q').$$

Proof. Suppose $(T_Q, D_Q) \restriction_U k = (T_{Q'}, D_{Q'}) \restriction_U k$. We will prove that $(T_{P⨾Q}, D_{P⨾Q}) \restriction_U k = (T_{P⨾Q'}, D_{P⨾Q'}) \restriction_U k$, from which $d_U(P \mathbin{⨾} Q, P \mathbin{⨾} Q') \leq d_U(Q, Q')$ follows immediately.

Let $t \in \mathsf{divergences}(P \mathbin{⨾} Q)$ and $\mathsf{length}_U(t) \leq k$. Similarly to the previous lemma, we consider the possible alternatives.

- Suppose $t \in \mathsf{divergences}(P)$. By definition, $t \in \mathsf{divergences}(P \mathbin{⨾} Q')$.
- Suppose $t = t_1 \frown t_2$ with $t_1 \frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(P)$ and $t_2 \in \mathsf{divergences}(Q)$. Let us observe that $\mathsf{length}_U(t_2) \leq \mathsf{length}_U(t) \leq k$. Since by assumption $D_Q \restriction_U k = D_{Q'} \restriction_U k$, $t_2 \in \mathsf{divergences}(Q')$. Then by definition, $t \in \mathsf{divergences}(P \mathbin{⨾} Q')$.

Let $t \in \mathsf{traces}_\perp(P \mathbin{⨾} Q)$ and $\mathsf{length}_U(t) \leq k$.

- Let first $t \in \mathsf{divergences}(P \mathbin{⨾} Q)$. We already proved that $t \in \mathsf{divergences}(P \mathbin{⨾} Q')$ and therefore, $t \in \mathsf{traces}_\perp(P \mathbin{⨾} Q')$.

- Let now $t \in \mathsf{traces}(P) \cap \Sigma^*$. Then by definition, $t \in \mathsf{traces}(P \,⨾\, Q') \subseteq \mathsf{traces}_\bot(P \,⨾\, Q')$.
- Let finally $t = t_1{}^\frown t_2$ with $t_1{}^\frown\langle\checkmark\rangle \in \mathsf{traces}(P) \subseteq \mathsf{traces}_\bot(P)$, $t_2 \in \mathsf{traces}(Q) \subseteq \mathsf{traces}_\bot(Q)$. Since $\mathsf{length}_U(t) \leq k$, $\mathsf{length}_U(t_2) \leq k$. Then, by assumption, $t_2 \in \mathsf{traces}_\bot(Q')$.
  - If $t_2 \in \mathsf{traces}(Q')$, by definition, $t = t_1{}^\frown t_2 \in \mathsf{traces}(P \,⨾\, Q') \subseteq \mathsf{traces}_\bot(P \,⨾\, Q')$.
  - Let $t_2 \in \mathsf{divergences}(Q')$. Since $t_1{}^\frown\langle\checkmark\rangle \in \mathsf{traces}_\bot(P)$, by definition, $t = t_1{}^\frown t_2 \in \mathsf{divergences}(P \,⨾\, Q') \subseteq \mathsf{traces}_\bot(P \,⨾\, Q')$.

Therefore, $D_{P⨾Q} \upharpoonright_U k \subseteq D_{P⨾Q'} \upharpoonright_U k$ and $T_{P⨾Q} \upharpoonright_U k \subseteq T_{P⨾Q'} \upharpoonright_U k$. The reverse containments are established similarly by symmetry. Therefore, $(T_{P⨾Q}, D_{P⨾Q}) \upharpoonright_U k = (T_{P⨾Q'}, D_{P⨾Q'}) \upharpoonright_U k$, and hence $d_U(P \,⨾\, Q, P \,⨾\, Q') \leq d_U(Q, Q')$. □

Lemma 4.5.11. *Let P, Q and Q' be CSP processes. Let P always communicate an event from $U \subseteq \Sigma$ before it does a $\checkmark$. Then,*

$$d_U(P \,⨾\, Q, P \,⨾\, Q') \leq \frac{1}{2} d_U(Q, Q').$$

Proof. Suppose $(T_Q, D_Q) \upharpoonright_U k = (T_{Q'}, D_{Q'}) \upharpoonright_U k$. We will prove that $(T_{P⨾Q}, D_{P⨾Q}) \upharpoonright_U k+1 = (T_{P⨾Q'}, D_{P⨾Q'}) \upharpoonright_U k+1$, which implies $d_U(P \,⨾\, Q, P \,⨾\, Q') \leq \frac{1}{2} d_U(Q, Q')$.

Let $t \in \mathsf{traces}_\bot(P \,⨾\, Q)$ and $\mathsf{length}_U(t) \leq k+1$.

- Suppose $t \in \mathsf{divergences}(P \,⨾\, Q)$.
  - If $t \in \mathsf{divergences}(P)$, by definition, $t \in \mathsf{divergences}(P \,⨾\, Q') \subseteq \mathsf{traces}_\bot(P \,⨾\, Q')$.
  - Let $t = t_1{}^\frown t_2$ with $t_1{}^\frown\langle\checkmark\rangle \in \mathsf{traces}_\bot(P)$, $t_2 \in \mathsf{divergences}(Q) \subseteq \mathsf{traces}_\bot(Q)$. Since P always communicates an event from $U \subseteq \Sigma$ before it can do a $\checkmark$, t_1 contains an event from U. Therefore, $\mathsf{length}_U(t_2) \leq k$. Then, since by assumption $(T_Q, D_Q) \upharpoonright_U k = (T_{Q'}, D_{Q'}) \upharpoonright_U k$, $t_2 \in \mathsf{divergences}(Q')$. Therefore, by definition, $t = t_1{}^\frown t_2 \in \mathsf{divergences}(P \,⨾\, Q') \subseteq \mathsf{traces}_\bot(P \,⨾\, Q')$.
- Suppose $t \in \mathsf{traces}_\bot(P \,⨾\, Q)$. Similarly to the previous lemmas, we consider the 3 possible alternatives.

- If $t \in \mathsf{divergences}(P \fatsemi Q)$, we already proved that $t \in \mathsf{divergences}(P \fatsemi Q') \subseteq \mathsf{traces}_\perp(P \fatsemi Q')$.
- If $t \in \mathsf{traces}(P) \cap \Sigma^*$, then by definition, $t \in \mathsf{traces}(P \fatsemi Q') \subseteq \mathsf{traces}_\perp(P \fatsemi Q')$.
- Let $t = t_1 {}^\frown t_2$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(P)$, $t_2 \in \mathsf{traces}(Q)$. Since P always communicates an event from $U \subseteq \Sigma$ before it does a $\checkmark$, t_1 contains an event from U. Therefore, $\mathsf{length}_U(t_2) \leq k$. Then, by assumption, $t_2 \in \mathsf{traces}_\perp(Q')$.
  * If $t_2 \in \mathsf{traces}(Q')$, by definition, $t = t_1 {}^\frown t_2 \in \mathsf{traces}(P \fatsemi Q') \subseteq \mathsf{traces}_\perp(P \fatsemi Q')$.
  * Let $t_2 \in \mathsf{divergences}(Q')$. Since $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(P)$, by definition, $t = t_1 {}^\frown t_2 \in \mathsf{divergences}(P \fatsemi Q') \subseteq \mathsf{traces}_\perp(P \fatsemi Q')$.

Therefore, $D_{P\fatsemi Q} \upharpoonright_U k+1 \subseteq D_{P\fatsemi Q'} \upharpoonright_U k+1$ and $T_{P\fatsemi Q} \upharpoonright_U k+1 \subseteq T_{P\fatsemi Q'} \upharpoonright_U k+1$. The reverse containments are established similarly by symmetry. Therefore, $(T_{P\fatsemi Q}, D_{P\fatsemi Q}) \upharpoonright_U k+1 = (T_{P\fatsemi Q'}, D_{P\fatsemi Q'}) \upharpoonright_U k+1$, and hence $d_U(P \fatsemi Q, P \fatsemi Q') \leq \frac{1}{2} d_U(Q, Q')$. □

Lemma 4.5.7 ($\sqcap$)**.** *For any CSP processes P, P' and Q,*

$$d_U(P \sqcap Q, P' \sqcap Q) \leq d_U(P, P').$$

Proof. Suppose $(T_P, D_P) \upharpoonright_U k = (T_{P'}, D_{P'}) \upharpoonright_U k$. We will prove that $(T_{P\sqcap Q}, D_{P\sqcap Q}) \upharpoonright_U k = (T_{P'\sqcap Q}, D_{P'\sqcap Q}) \upharpoonright_U k$, which directly implies $d_U(P \sqcap Q, P' \sqcap Q) \leq d_U(P, P')$.

Let $t \in \mathsf{divergences}(P \sqcap Q)$ and $\mathsf{length}_U(t) \leq k$.

- Suppose $t \in \mathsf{divergences}(P)$. By assumption, $D_P \upharpoonright_U k = D_{P'} \upharpoonright_U k$. Therefore, $t \in \mathsf{divergences}(P') \subseteq \mathsf{divergences}(P' \sqcap Q)$.
- Suppose $t \in \mathsf{divergences}(Q)$. By definition, $t \in \mathsf{divergences}(P' \sqcap Q)$.

Let $t \in \mathsf{traces}_\perp(P \sqcap Q)$ and $\mathsf{length}_U(t) \leq k$. We have that $\mathsf{traces}_\perp(P \sqcap Q) = \mathsf{traces}(P \sqcap Q) \cup \mathsf{divergences}(P \sqcap Q) = \mathsf{traces}(P) \cup \mathsf{divergences}(P) \cup \mathsf{traces}(Q) \cup \mathsf{divergences}(Q) = \mathsf{traces}_\perp(P) \cup \mathsf{traces}_\perp(Q)$.

- Let $t \in \mathsf{traces}_\perp(P)$. By assumption, $T_P \upharpoonright_U k = T_{P'} \upharpoonright_U k$. Therefore, $t \in \mathsf{traces}_\perp(P') \subseteq \mathsf{traces}_\perp(P' \sqcap Q)$.
- Let $t \in \mathsf{traces}_\perp(Q)$. By definition, $t \in \mathsf{traces}_\perp(P' \sqcap Q)$.

Therefore, $D_{P \sqcap Q} \upharpoonright_U k \subseteq D_{P' \sqcap Q} \upharpoonright_U k$ and $T_{P \sqcap Q} \upharpoonright_U k \subseteq T_{P' \sqcap Q} \upharpoonright_U k$. The reverse containments are established similarly by symmetry. Therefore, $(T_{P \sqcap Q}, D_{P \sqcap Q}) \upharpoonright_U k = (T_{P' \sqcap Q}, D_{P' \sqcap Q}) \upharpoonright_U k$, and hence $d_U(P \sqcap Q, P' \sqcap Q) \leq d_U(P, P')$. □

Lemma 4.5.7 ($\Box$). *For any CSP processes P, P' and Q,*

$$d_U(P \Box Q, P' \Box Q) \leq d_U(P, P').$$

Proof. Same as for $\sqcap$. □

Lemma 4.5.7 ($\underset{A}{\parallel}$). *For any CSP processes P, P' and Q, and any $A \subseteq \Sigma$,*

$$d_U(P \underset{A}{\parallel} Q, P' \underset{A}{\parallel} Q) \leq d_U(P, P').$$

Proof. Suppose $(T_P, D_P) \upharpoonright_U k = (T_{P'}, D_{P'}) \upharpoonright_U k$. We will prove that $(T_{P \underset{A}{\parallel} Q}, D_{P \underset{A}{\parallel} Q}) \upharpoonright_U k = (T_{P' \underset{A}{\parallel} Q}, D_{P' \underset{A}{\parallel} Q}) \upharpoonright_U k$, which directly implies $d_U(P \underset{A}{\parallel} Q, P' \underset{A}{\parallel} Q) \leq d_U(P, P')$.

Let $t \in \mathsf{divergences}(P \underset{A}{\parallel} Q)$ and $\mathsf{length}_U(t) \leq k$. Therefore, $t = u^\frown v$ with $u \in (s \underset{A}{\parallel} r \cap \Sigma^*)$, where $s \in \mathsf{traces}_\perp(P)$, $r \in \mathsf{traces}_\perp(Q)$ and, $s \in \mathsf{divergences}(P)$ or $r \in \mathsf{divergences}(Q)$. Let us recall that v ranges over $\Sigma^{*\checkmark}$, in accordance with Axiom 4. Let us further observe that $\mathsf{length}_U(s) \leq \mathsf{length}_U(u) \leq \mathsf{length}_U(t) \leq k$. Therefore, by assumption, $s \in \mathsf{traces}_\perp(P')$.

- Let $s \in \mathsf{divergences}(P)$. By assumption, $s \in \mathsf{divergences}(P')$. Therefore by definition, $t \in \mathsf{divergences}(P' \underset{A}{\parallel} Q)$.
- Let $r \in \mathsf{divergences}(Q)$. Since $s \in \mathsf{traces}_\perp(P')$, by definition, $t \in \mathsf{divergences}(P' \underset{A}{\parallel} Q)$.

Let $t \in \mathsf{traces}_\perp(P \underset{A}{\parallel} Q)$ and $\mathsf{length}_U(t) \leq k$.

- Suppose $t \in \mathsf{divergences}(P \underset{A}{\parallel} Q)$. We already proved that $t \in \mathsf{divergences}(P \underset{A}{\parallel} Q') \subseteq \mathsf{traces}_\perp(P' \underset{A}{\parallel} Q)$.

- Suppose $t \in \mathsf{traces}(P \underset{A}{\|} Q)$. Therefore, there exist $s \in \mathsf{traces}(P) \subseteq \mathsf{traces}_\perp(P)$, $r \in \mathsf{traces}(Q) \subseteq \mathsf{traces}_\perp(Q)$, such that $t \in s \underset{A}{\|} r$. By assumption, $s \in \mathsf{traces}_\perp(P')$.
  - If $s \in \mathsf{traces}(P')$, by definition, $t \in \mathsf{traces}(P' \underset{A}{\|} Q) \subseteq \mathsf{traces}_\perp(P' \underset{A}{\|} Q)$.
  - If $s \in \mathsf{divergences}(P')$, by definition, $t \in \mathsf{divergences}(P' \underset{A}{\|} Q) \subseteq \mathsf{traces}_\perp(P' \underset{A}{\|} Q)$.

Therefore, $D_{P\underset{A}{\|}Q} \restriction_U k \subseteq D_{P'\underset{A}{\|}Q} \restriction_U k$ and $T_{P\underset{A}{\|}Q} \restriction_U k \subseteq T_{P'\underset{A}{\|}Q} \restriction_U k$. The reverse containments are established similarly by symmetry. Therefore, $(T_{P\underset{A}{\|}Q}, D_{P\underset{A}{\|}Q}) \restriction_U k = (T_{P'\underset{A}{\|}Q}, D_{P'\underset{A}{\|}Q}) \restriction_U k$, and hence $d_U(P \underset{A}{\|} Q, P' \underset{A}{\|} Q) \leq d_U(P, P')$.

□

Lemma 4.5.9. *Let P and Q be CSP processes and let $A \subseteq \Sigma$ satisfy $A \cap U = \emptyset$. Then,*

$$d_U(P \setminus A, Q \setminus A) \leq d_U(P, Q).$$

Proof. Suppose $(T_P, D_P) \restriction_U k = (T_Q, D_Q) \restriction_U k$. We will prove that $(T_{P\setminus A}, D_{P\setminus A}) \restriction_U k = (T_{Q\setminus A}, D_{Q\setminus A}) \restriction_U k$, which implies $d_U(P \setminus A, Q \setminus A) \leq d_U(P, Q)$.

Let $t \in \mathsf{divergences}(P \setminus A)$ and $\mathsf{length}_U(t) \leq k$. We consider the possible alternatives for t.

- Suppose that there exists $s \in \mathsf{divergences}(P)$, such that $t = (s \restriction (\Sigma \setminus A))^\frown r$. Let us recall that r ranges over $\Sigma^{*\checkmark}$, in accordance with Axiom 4. Since $A \cap U = \emptyset$, $\mathsf{length}_U(s) = \mathsf{length}_U(s \restriction (\Sigma \setminus A)) \leq \mathsf{length}_U(t) \leq k$. Then, by assumption, $s \in \mathsf{divergences}(Q)$. Therefore, by definition, $t \in \mathsf{divergences}(Q \setminus A)$.

- Now suppose that there exists $u \in \Sigma^\omega$, such that $u \restriction (\Sigma \setminus A)$ is finite, for each $s < u$, $s \in \mathsf{traces}_\perp(P)$, and $t = u \restriction (\Sigma \setminus A) {}^\frown r$ (where r ranges over $\Sigma^{*\checkmark}$, in accordance with Axiom 4). Since $A \cap U = \emptyset$, $\mathsf{length}_U(u) = \mathsf{length}_U(u \restriction (\Sigma \setminus A)) \leq \mathsf{length}_V(t) \leq k$. Then, by assumption, for each $s < u$, $s \in \mathsf{traces}_\perp(Q)$. Then $t \in \mathsf{divergences}(Q \setminus A)$ follows by definition.

Let $t \in \mathsf{traces}_\perp(P \setminus A)$ and $\mathsf{length}_U(t) \leq k$.

- Let first $t \in \mathsf{divergences}(P \setminus A)$. We already proved that $t \in \mathsf{divergences}(Q \setminus A) \subseteq \mathsf{traces}_\perp(Q \setminus A)$.

- Let now $t \in \mathsf{traces}(P \setminus A)$. Therefore, there exists $s \in \mathsf{traces}(P) \subseteq \mathsf{traces}_\perp(P)$, such that $t = s \upharpoonright (\Sigma \backslash A)$. From $A \cap U = \emptyset$, $\mathsf{length}_U(s) = \mathsf{length}_U(s \upharpoonright (\Sigma \backslash A)) = \mathsf{length}_U(t) \leq k$. Then, by assumption, $s \in \mathsf{traces}_\perp(Q)$.
  - If $s \in \mathsf{divergences}(Q)$, by definition, $t \in \mathsf{divergences}(Q \setminus A) \subseteq \mathsf{traces}_\perp(Q \setminus A)$.
  - If $s \in \mathsf{traces}(Q)$, by definition, $t \in \mathsf{traces}(Q \setminus A) \subseteq \mathsf{traces}_\perp(Q \setminus A)$.

Therefore, $D_{P\backslash A} \upharpoonright_U k \subseteq D_{Q\backslash A} \upharpoonright_U k$ and $T_{P\backslash A} \upharpoonright_U k \subseteq T_{Q\backslash A} \upharpoonright_U k$. The reverse containments are established similarly by symmetry. Therefore, $(T_{P\backslash A}, D_{P\backslash A}) \upharpoonright_U k = (T_{Q\backslash A}, D_{Q\backslash A}) \upharpoonright_U k$, and hence $d_U(P \setminus A, Q \setminus A) \leq d_U(P, Q)$. □

Lemma 4.5.10. *Let P and Q be CSP processes, $R \subseteq \Sigma \times \Sigma$ be a renaming relation on Σ, and $R(U) = \{y \mid \exists\, x \in U \bullet x\ R\ y\}$. Then,*

$$d_{R(U)}(P[\![R]\!], Q[\![R]\!]) \leq d_U(P, Q).$$

Proof. Suppose $(T_P, D_P) \upharpoonright_U k = (T_Q, D_Q) \upharpoonright_U k$. We will prove that $(T_{P[\![R]\!]}, D_{P[\![R]\!]}) \upharpoonright_{R(U)} k = (T_{Q[\![R]\!]}, D_{Q[\![R]\!]}) \upharpoonright_{R(U)} k$.

Let $t \in \mathsf{divergences}(P[\![R]\!])$ and $\mathsf{length}_{R(U)}(t) \leq k$. Then there exist $s_1, t_1 \in \Sigma^*, r \in \Sigma^{*\checkmark}$, such that $s_1 \in \mathsf{divergences}(P) \cap \Sigma^*$, $s_1\ R\ t_1$ and $t = t_1 {}^\frown r$ (where r ranges over $\Sigma^{*\checkmark}$, in accordance with Axiom 4). Then, $\mathsf{length}(s_1) = \mathsf{length}(t_1)$ and for $1 \leq i \leq \mathsf{length}(s_1)$, $s_{1_i}\ R\ t_{1_i}$. Therefore, $\mathsf{length}_U(s_1) = \mathsf{length}_{R(U)}(t_1) \leq \mathsf{length}_{R(U)}(t) \leq k$ and, by assumption, $s_1 \in \mathsf{divergences}(Q) \cap \Sigma^*$. Hence, by definition, $t \in \mathsf{divergences}(Q[\![R]\!])$.

Let $t \in \mathsf{traces}_\perp(P[\![R]\!])$ and $\mathsf{length}_{R(U)}(t) \leq k$.

- If $t \in \mathsf{divergences}(P[\![R]\!])$, we already proved that $t \in \mathsf{divergences}(Q[\![R]\!]) \subseteq \mathsf{traces}_\perp(Q[\![R]\!])$.
- Let $t \in \mathsf{traces}(P[\![R]\!])$. Then there exists $s \in \mathsf{traces}(P)$, such that $s\ R\ t$. Therefore, $\mathsf{length}_U(s) = \mathsf{length}_{R(U)}(t) \leq k$ and, by assumption, $s \in \mathsf{traces}_\perp(Q)$.
  - If $s \in \mathsf{traces}(Q)$, by definition, $t \in \mathsf{traces}(Q[\![R]\!]) \subseteq \mathsf{traces}_\perp(Q[\![R]\!])$.
  - If $s \in \mathsf{divergences}(Q)$, by definition, $t \in \mathsf{divergences}(Q[\![R]\!]) \subseteq \mathsf{traces}_\perp(Q[\![R]\!])$.

Therefore, $D_{P[\![R]\!]} \restriction_{R(U)} k \subseteq D_{Q[\![R]\!]} \restriction_{R(U)} k$ and $T_{P[\![R]\!]} \restriction_{R(U)} k \subseteq T_{Q[\![R]\!]} \restriction_{R(U)} k$. The reverse containments are established similarly by symmetry. Therefore, $(T_{P[\![R]\!]}, D_{P[\![R]\!]}) \restriction_{R(U)} k = (T_{Q[\![R]\!]}, D_{Q[\![R]\!]}) \restriction_{R(U)} k$, and hence $d_{R(U)}(P[\![R]\!], Q[\![R]\!]) \leq d_U(P, Q)$. $\square$

A.4 Proofs for Section 4.6

Proposition 4.6.2. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a CSP term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $N_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.11. If $(U, V) \in N_X(P)$, then for all $T_1, T_2, \Theta_1, \ldots, \Theta_n \in \mathcal{T}^{\Downarrow}$, $d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) \leq d_U(T_1, T_2)$.*

Proof. Structural induction on P. Let us take arbitrary $T_1, T_2, \Theta_1, \ldots, \Theta_n \in \mathcal{T}^{\Downarrow}$.

- $\mathsf{N}_X(P) = \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ whenever X is not free in P

  Proof. Let $(U, V) \in \mathsf{N}_X(P)$. Then $d_U(T_1, T_2) \geq 0 = d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) = d_V(P(\overline{\Theta}), P(\overline{\Theta}))$. □

- $\mathsf{N}_X(a \longrightarrow P) = \mathsf{N}_X(P)$

  Proof. Suppose $(U, V) \in \mathsf{N}_X(a \longrightarrow P)$. By construction, $(U, V) \in \mathsf{N}_X(P)$. Then:

  $$\begin{aligned} d_U(T_1, T_2) &\geq d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) && \text{// induction hypothesis} \\ &\geq d_V(a \longrightarrow P(T_1, \overline{\Theta}), a \longrightarrow P(T_2, \overline{\Theta})) && \text{// Lemma 4.5.8} \\ &= d_V((a \longrightarrow P)(T_1, \overline{\Theta}), (a \longrightarrow P)(T_2, \overline{\Theta})) \end{aligned}$$ □

- $\mathsf{N}_X(P \setminus A) = \{(U, V) \mid (U, V') \in \mathsf{N}_X(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\}$

  Proof. Suppose $(U, V) \in \mathsf{N}_X(P \setminus A)$. By construction, there exists V', such that $(U, V') \in \mathsf{N}_X(P)$, $V' \subseteq V$ and $V' \cap A = \emptyset$. We will prove that for any $T_1, T_2 \in \mathcal{T}^{\Downarrow}$, $d_U(T_1, T_2) \geq d_V(P(T_1, \overline{\Theta}) \setminus A, P(T_2, \overline{\Theta}) \setminus A)$.

  $$\begin{aligned} d_U(T_1, T_2) &\geq d_{V'}(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) && \text{// induction hypothesis} \\ &\geq d_{V'}(P(T_1, \overline{\Theta}) \setminus A, P(T_2, \overline{\Theta}) \setminus A) && \text{// } V' \cap A = \emptyset\text{, Lemma 4.5.9} \\ &\geq d_V(P(T_1, \overline{\Theta}) \setminus A, P(T_2, \overline{\Theta}) \setminus A) && \text{// } V' \subseteq V,\ U \mapsto d_U \text{ antitone} \end{aligned}$$ □

- $\mathsf{N}_X(P_1 \oplus P_2) = \mathsf{N}_X(P_1) \cap \mathsf{N}_X(P_2) = \{(U_1 \cap U_2, V_1 \cup V_2) \mid (U_i, V_i) \in \mathsf{N}_X(P_i)\}$ for $\oplus \in \{\sqcap, \Box, ;, \underset{A}{\|}\}$.

Proof. Suppose $(U,V) \in \mathsf{N}_X(P_1 \oplus P_2)$. By construction, there exist $(U_1,V_1) \in \mathsf{N}_X(P_1)$ and $(U_2,V_2) \in \mathsf{N}_X(P_2)$, such that $U = U_1 \cap U_2$ and $V = V_1 \cup V_2$. Therefore, $(U,V) \in \mathsf{N}_X(P_1)$, $(U,V) \in \mathsf{N}_X(P_2)$ (antitoneness).

$$\begin{aligned}
&d_V((P_1 \oplus P_2)(T_1,\overline{\Theta}),(P_1 \oplus P_2)(T_2,\overline{\Theta}))\\
&= d_V(P_1(T_1,\overline{\Theta}) \oplus P_2(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta}) \oplus P_2(T_2,\overline{\Theta}))\\
&\text{// ultrametric inequality}\\
&\leq max\{\ d_V(P_1(T_1,\overline{\Theta}) \oplus P_2(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta}) \oplus P_2(T_1,\overline{\Theta})),\\
&\qquad\qquad d_V(P_1(T_2,\overline{\Theta}) \oplus P_2(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta}) \oplus P_2(T_2,\overline{\Theta}))\}\\
&\text{// Lemma 4.5.7}\\
&\leq max\{\ d_V(P_1(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta})) \quad \text{// } \leq d_U(T_1,T_2) \text{ by induction hypothesis for } P_1\\
&\qquad\qquad d_V(P_2(T_1,\overline{\Theta}), P_2(T_2,\overline{\Theta}))\} \text{ // } \leq d_U(T_1,T_2) \text{ by induction hypothesis for } P_2\\
&\leq d_U(T_1,T_2)
\end{aligned}$$
□

- $\mathsf{N}_X(P[\![R]\!]) = \{(U,V) \mid (U,V') \in \mathsf{N}_X(P) \wedge R(V') \subseteq V\}$

Proof. Suppose $(U,V) \in \mathsf{N}_X(P[\![R]\!])$. By construction, there exists V', such that $(U,V') \in \mathsf{N}_X(P)$ and $R(V') \subseteq V$.

$$\begin{aligned}
d_U(T_1,T_2) &\geq d_{V'}(P(T_1,\overline{\Theta}), P(T_2,\overline{\Theta})) && \text{// induction hypothesis}\\
&\geq d_{R(V')}(P(T_1,\overline{\Theta})[\![R]\!], P(T_2,\overline{\Theta})[\![R]\!]) && \text{// Lemma 4.5.10}\\
&\geq d_V(P(T_1,\overline{\Theta})[\![R]\!], P(T_2,\overline{\Theta})[\![R]\!]) && \text{// } R(V') \subseteq V,\ U \mapsto d_U \text{ antitone}
\end{aligned}$$
□

- $\mathsf{N}_X(X) = \{(U,V) \mid U \subseteq V\}$

Proof.

$$\begin{aligned}
d_U(T_1,T_2) &\geq d_V(T_1,T_2) && \text{// } U \subseteq V,\ U \mapsto d_U \text{ antitone}\\
&= d_V((X)(T_1,\overline{\Theta}),(X)(T_2,\overline{\Theta}))
\end{aligned}$$
□

- $\mathsf{N}_X(\mu Y \,\text{.}\, P) = \{(U,V) \mid (U',V') \in \mathsf{N}_X(P) \wedge (V',V') \in \mathsf{N}_Y(P) \wedge U \subseteq U' \wedge V' \subseteq V\}$ if $Y \neq X$

Proof. Suppose $(U,V) \in \mathsf{N}_X(\mu Y \,\text{.}\, P)$ for $X \neq Y$ and X, Y free in $P(X,Y,Z_1,\ldots,Z_n)$. By construction, there exist $U_X, V_X \subseteq \Sigma$ such that:

1. $(U_X,V_X) \in \mathsf{N}_X(P)$
2. $U \subseteq U_X$, $V_X \subseteq V$

3. $(V_X, V_X) \in \mathsf{N}_Y(P)$

Therefore, by induction hypothesis, we have the following for all $T_1, T_2, \xi, \overline{\Theta} \in \mathcal{T}^{\Downarrow}$:

$$d_{U_X}(T_1, T_2) \geq d_{V_X}(P(T_1, \xi, \overline{\Theta}), P(T_2, \xi, \overline{\Theta})) \tag{A.1}$$

$$d_{V_X}(T_1, T_2) \geq d_{V_X}(P(\xi, T_1, \overline{\Theta}), P(\xi, T_2, \overline{\Theta})) \tag{A.2}$$

$$\begin{aligned} d_U(T_1, T_2) &\geq d_{U_X}(T_1, T_2) && // \ U \subseteq U_X, \text{ antitoneness} \\ &\geq d_{V_X}(P(T_1, \xi, \overline{\Theta}), P(T_2, \xi, \overline{\Theta})) && // \text{ from A.1} \end{aligned}$$

Let $P_1(Y) = P(T_1, Y, \overline{\Theta})$, $P_2(Y) = P(T_2, Y, \overline{\Theta})$. $P_1(Y)$ and $P_2(Y)$ are continuous over $\sqsubseteq$. Therefore, there exist $\mu Y \centerdot P_1(Y) = \bigcap_{n=0}^{\infty} P_1^n = P_1^*$ and $\mu Y \centerdot P_2(Y) = \bigcap_{n=0}^{\infty} P_2^n = P_2^*$, where for $i = 1, 2$, $P_i^0 = \bot = DIV$, $P_i^{n+1} = P_i(P_i^n)$.

We will prove by induction that

$$d_{V_X}(P_1^n, P_2^n) \leq d_{U_X}(T_1, T_2) \text{ for } n \geq 1. \tag{A.3}$$

– Let $n = 1$.
$$\begin{aligned} d_{U_X}(T_1, T_2) &\geq d_{V_X}(P(T_1, DIV, \overline{\Theta}), P(T_2, DIV, \overline{\Theta})) \quad // \text{ from A.1} \\ &= d_{V_X}(P_1^1, P_2^1) \end{aligned}$$
– Suppose $d_{V_X}(P_1^n, P_2^n) \leq d_{U_X}(T_1, T_2)$.
$$\begin{aligned} d_{V_X}(P_1^{n+1}, P_2^{n+1}) &= d_{V_X}(P(T_1, P_1^n, \overline{\Theta}), P(T_2, P_2^n, \overline{\Theta})) \\ & // \text{ ultrametric inequality} \\ &\leq max\{d_{V_X}(P(T_1, P_1^n, \overline{\Theta}), P(T_2, P_1^n, \overline{\Theta})), \\ &\qquad d_{V_X}(P(T_2, P_1^n, \overline{\Theta}), P(T_2, P_2^n, \overline{\Theta}))\} \\ &\leq max\{d_{U_X}(T_1, T_2), && // \text{ from A.1} \\ &\qquad d_{V_X}(P_1^n, P_2^n)\} && // \text{ from A.2} \\ &\leq max\{d_{U_X}(T_1, T_2), \\ &\qquad d_{U_X}(T_1, T_2\} && // \text{ from A.3, local i.h.} \\ &\leq d_{U_X}(T_1, T_2) \end{aligned}$$

Let $d_{U_X}(T_1, T_2) = 2^{-k}$ for some $k \in \mathbb{N}$. Now suppose for the sake of the argument that $d_{V_X}(P_1^*, P_2^*) > d_{U_X}(T_1, T_2) = 2^{-k}$, and let without loss of generality P_1^* and P_2^* differ on the sets of their divergences. Therefore, again without loss of generality, there exists $s \in \mathsf{divergences}(P_1^*)$ such that $s \notin \mathsf{divergences}(P_2^*)$ and $\mathsf{length}_{V_X}(s) \leq k$. Then, since $P_i^* = \bigcap_{n=0}^{\infty} P_i^n$, $s \in P_1^n$ for all $n \in \mathbb{N}$, but there exists $l \in \mathbb{N}$ such that $s \notin P_2^l$. But then $d_{V_X}(P_1^l, P_2^l) > 2^{-k} = d_{U_X}(T_1, T_2)$, which is a contradiction with

A.3. Therefore, $d_{V_X}(P_1^*, P_2^*) \leq d_{U_X}(T_1, T_2)$. Then, since $U \subseteq U_X$ and $V_X \subseteq V$, by antitoneness, $d_V((\mu Y \centerdot P)(T_1, \overline{\Theta}), (\mu Y \centerdot P)(T_2, \overline{\Theta})) = d_V(P_1^*, P_2^*) \leq d_U(T_1, T_2)$.

□

□

Proposition A.4.1. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a CSP term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $\mathsf{G} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$, $\mathsf{C}_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ and $\mathsf{F} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figures 4.12, 4.13 and 4.14, respectively. Then:*

1. *If $V \in \mathsf{G}(P)$, then, with any processes substituted for the free variables of P (and in particular DIV), P must communicate an event from V before it can do a $\checkmark$.*
2. *If $(U, V) \in \mathsf{C}_X(P)$, then for all processes $T_1, T_2, \Theta_1, \ldots, \Theta_n \in \mathcal{T}^{\Downarrow}$, $d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta})) \leq \frac{1}{2} d_U(T_1, T_2)$.*
3. *If $(U, V) \in \mathsf{F}(P)$, then, for any collection of U-fair livelock-free processes $\theta_0, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, the process $P(\theta_0, \ldots, \theta_n)$ is livelock-free and V-fair.*

Proof. We carry out the proof by induction on the structure of P. For clarity, we prove (1), (2) and (3) one by one, in Propositions 4.6.3, 4.6.4 and 4.6.6, respectively. In each of these propositions, our induction hypothesis is that at any point all (1), (2) and (3) hold for any subterm of P. □

Proposition 4.6.3. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a CSP term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $\mathsf{G} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.12. If $V \in \mathsf{G}(P)$, then, with any processes substituted for the free variables of P (and in particular DIV), P must communicate an event from V before it can do a $\checkmark$.*

Proof. Structural induction on P. We will write $\widehat{P}$ to denote the result of substituting all free variables in P with the most general process $\perp = DIV$. For each process ξ, $DIV \sqsubseteq \xi$. Therefore, by monotonicity of CSP operators (see Section 2.2.3), for any process term $C(X)$, $C(DIV) \sqsubseteq C(\xi)$.

- $\mathsf{G}(STOP) = \mathcal{P}(\Sigma)$.

  Proof. $STOP$ cannot terminate and therefore the property holds vacuously. $\square$

- $\mathsf{G}(a \longrightarrow P) = \mathsf{G}(P) \cup \{V \mid a \in V\}$.

  Proof. Let $V \in \mathsf{G}(a \longrightarrow P)$ and $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{a \longrightarrow P}) = \mathsf{traces}_\perp(a \longrightarrow \widehat{P})$. Therefore $t = \langle a\rangle^\frown r^\frown\langle\checkmark\rangle$ for some $r \in \Sigma^*$ such that $r^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{P})$ and $s = \langle a\rangle^\frown r$. Since $V \in \mathsf{G}(a \longrightarrow P)$, by construction, $V \in \mathsf{G}(P)$ or $a \in V$.

  - Suppose $V \in \mathsf{G}(P)$. Then by induction hypothesis, $r^\frown\langle\checkmark\rangle$ contains an event from V, and therefore so do s and t.
  - Suppose $a \in V$. Then $t = \langle a\rangle^\frown r^\frown\langle\checkmark\rangle$ contains the event $a \in V$ before $\checkmark$.

  $\square$

- $\mathsf{G}(SKIP) = \emptyset$. $\square$

- $\mathsf{G}(P_1 \oplus P_2) = \mathsf{G}(P_1) \cap \mathsf{G}(P_2)$ for $\oplus \in \{\sqcap, \Box\}$.

  Proof. Let $V \in \mathsf{G}(P_1 \oplus P_2)$ and $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{P_1 \oplus P_2}) = \mathsf{traces}_\perp(\widehat{P_1}) \cup \mathsf{traces}_\perp(\widehat{P_2})$. Therefore $t \in \mathsf{traces}_\perp(\widehat{P_1})$ or $t \in \mathsf{traces}_\perp(\widehat{P_2})$. Let, without loss of generality, $t \in \mathsf{traces}_\perp(\widehat{P_1})$. By construction, $V \in \mathsf{G}(P_1)$. Then by induction hypothesis, s contains an event from V. $\square$

- $\mathsf{G}(P_1 \mathbin{;} P_2) = \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if } P_1 \text{ is closed and } \mathsf{F}(P_1) \neq \emptyset \\ \mathsf{G}(P_1) & \text{otherwise} \end{cases}$

  Proof. Let $V \in \mathsf{G}(P_1 \mathbin{;} P_2)$ and $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{P_1 \mathbin{;} P_2})$.

  Let first P_1 be closed and $\mathsf{F}(P_1) \neq \emptyset$. Then by Proposition A.4.1 (3), P_1 is livelock-free, and therefore $\mathsf{divergences}(P_1) = \emptyset$. Therefore, $t = t_1{}^\frown t_2{}^\frown\langle\checkmark\rangle$ with $t_1{}^\frown\langle\checkmark\rangle \in \mathsf{traces}(\widehat{P_1})$ and $t_2{}^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{P_2})$. In this case by construction $V \in \mathsf{G}(P_1) \cup \mathsf{G}(P_2)$. Let without loss of generality $V \in \mathsf{G}(P_1)$. Then by induction hypothesis, t_1 contains an event from V, and therefore so does t.

Let now P_1 be open or $\mathsf{F}(P_1) = \emptyset$. Then by construction $V \in \mathsf{G}(P_1)$. We consider the two possibilities for t.

- Let first $t = t_1 {}^\frown t_2 {}^\frown \langle \checkmark \rangle$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(\widehat{P_1})$ and $t_2 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(\widehat{P_2})$. Since $V \in \mathsf{G}(P_1)$, by induction hypothesis t_1 contains an event from V, and therefore so does t.
- Let now $t \in \mathsf{divergences}(P_1)$ and therefore, $t \in \mathsf{traces}_\perp(P_1)$. Then again, by induction hypothesis, t_1 contains an event from V, and therefore so does t.

□

- $\mathsf{G}(P_1 \underset{A}{\parallel} P_2) = \begin{cases} \mathsf{G}(P_1) \cup \mathsf{G}(P_2) & \text{if, for } i = 1,2,\ P_i \text{ is closed and } \mathsf{F}(P_i) \neq \emptyset \\ \mathsf{G}(P_1) \cap \mathsf{G}(P_2) & \text{otherwise} \end{cases}$

Proof. Let $V \in \mathsf{G}(P_1 \underset{A}{\parallel} P_2)$ and $t = s {}^\frown \langle \checkmark \rangle \in \mathsf{traces}_\perp(\widehat{P_1 \underset{A}{\parallel} P_2})$.

Let first both P_1 and P_2 be closed, $\mathsf{F}(P_1) \neq \emptyset$ and $\mathsf{F}(P_2) \neq \emptyset$. Then, by Proposition A.4.1 (3), P_1 and P_2 are livelock-free, and therefore $P_1 \underset{A}{\parallel} P_2$ is livelock-free. Therefore, $\mathsf{divergences}(\widehat{P_1 \underset{A}{\parallel} P_2}) = \emptyset$ and $\mathsf{traces}_\perp(\widehat{P_1 \underset{A}{\parallel} P_2}) = \mathsf{traces}(\widehat{P_1 \underset{A}{\parallel} P_2})$. By construction, $V \in \mathsf{G}(P_1) \cup \mathsf{G}(P_2)$. Let without loss of generality $V \in \mathsf{G}(P_2)$. Since $t \in \mathsf{traces}(\widehat{P_1 \underset{A}{\parallel} P_2})$, then, due to distributed termination, there exist t_1, t_2, such that $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(\widehat{P_1})$, $t_2 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(\widehat{P_2})$ and $t \in t_1 \underset{A}{\parallel} t_1$. By induction hypothesis, t_2 contains an event from V and therefore so does t.

Otherwise, $t \in \mathsf{traces}(\widehat{P_1 \underset{A}{\parallel} P_2})$ or $t \in \mathsf{divergences}(\widehat{P_1 \underset{A}{\parallel} P_2})$. We consider both alternatives. By construction, $V \in \mathsf{G}(P_1) \cap \mathsf{G}(P_2)$, i.e., $V \in \mathsf{G}(P_1)$ and $V \in \mathsf{G}(P_2)$.

- Let $t = s {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(\widehat{P_1 \underset{A}{\parallel} P_2})$. Then, due to distributed termination, there exist t_1, t_2, such that $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(\widehat{P_1})$, $t_2 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(\widehat{P_2})$ and $t \in t_1 \underset{A}{\parallel} t_1$. By induction hypothesis, both t_1 and t_2 contain an event from V and therefore so does t.
- Let $t = s {}^\frown \langle \checkmark \rangle \in \mathsf{divergences}(\widehat{P_1 \underset{A}{\parallel} P_2})$. Therefore, there exist s_1, s_2, t_1, t_2, such that $t_1 \in \mathsf{traces}_\perp(\widehat{P_1})$, $t_2 \in \mathsf{traces}_\perp(\widehat{P_2})$, $s_1 \in (t_1 \underset{A}{\parallel} t_2) \cap \Sigma^*$, $t = s_1 {}^\frown s_2 {}^\frown \langle \checkmark \rangle$, and $t_1 \in \mathsf{divergences}(\widehat{P_1})$ or $t_2 \in \mathsf{divergences}(\widehat{P_2})$. Let without loss of generality

$t_1 \in \mathsf{divergences}(\widehat{P_1})$. Then $t_1 \in \Sigma^*$, and by Axiom 4, $t_1{}^\frown\langle\checkmark\rangle \in \mathsf{divergences}(\widehat{P_1})$. Since $V \in \mathsf{G}(P_1)$, by induction hypothesis t_1 contains an event from V and therefore so does t.

□

- $\mathsf{G}(P[\![R]\!]) = \{V \mid V' \in \mathsf{G}(P) \wedge R(V') \subseteq V\}$.

Proof. Let $V \in \mathsf{G}(P[\![R]\!])$. Then, by construction, there exists $V' \in \mathsf{G}(P)$ with $R(V') \subseteq V$. Let $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{P[\![R]\!]})$. Then, $t \in \mathsf{divergences}(\widehat{P[\![R]\!]})$ or $t \in \mathsf{traces}(\widehat{P[\![R]\!]})$. We consider both alternatives.

- Suppose $t = s^\frown\langle\checkmark\rangle \in \mathsf{divergences}(\widehat{P[\![R]\!]})$. Therefore, there exist $s_1, s_2, r_1 \in \Sigma^*$, such that $r_1 \in \mathsf{divergences}(P)$, $r_1\ R\ s_1$ and $t = s_1{}^\frown s_2{}^\frown\langle\checkmark\rangle$. As $r_1 \in \mathsf{divergences}(P)$, by Axiom 4 we have $r_1{}^\frown\langle\checkmark\rangle \in \mathsf{divergences}(P)$. Then, by induction hypothesis for P, r_1 contains an event from V'. Since $r_1\ R\ s_1$, s_1 contains an event from $R(V') \subseteq V$. Therefore, since $t = s_1{}^\frown s_2{}^\frown\langle\checkmark\rangle$, t contains an event from V.
- Suppose $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}(\widehat{P[\![R]\!]})$. Therefore, there exist $t', s' \in \mathsf{traces}(\widehat{P})$, such that $t' = s'^\frown\langle\checkmark\rangle$ and $s'\ R\ s$. By induction hypothesis for P and t', s' contains an event from V'. Since $s'\ R\ s$, s contains an event from $R(V') \subseteq V$, and hence t contains an event from V.

□

- $$\mathsf{G}(P \setminus A) = \begin{cases} \{V \mid V' \in \mathsf{G}(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\} & \text{if } P \text{ is closed and} \\ & (\emptyset, \Sigma - A) \in \mathsf{F}(P) \\ \emptyset & \text{otherwise} \end{cases}$$

Proof. Let $V \in \mathsf{G}(P \setminus A)$. Let furthermore P be closed and $(\emptyset, \Sigma - A) \in \mathsf{F}(P)$. Then P does not have free process variables and by Proposition A.4.1 (3) we can conclude the following:

1. P is livelock-free, i.e., $\mathsf{divergences}(P) = \emptyset$ and $\mathsf{traces}_\perp(P) = \mathsf{traces}(P)$.
2. Any infinite trace u of P contains infinitely many events from $\Sigma - A$. Therefore $u \upharpoonright (\Sigma \setminus A)$ is infinite.

Let $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{P} \setminus A) = \mathsf{traces}_\perp(P \setminus A)$. Then either $t \in \mathsf{divergences}(P \setminus A)$ or $t \in \mathsf{traces}(P \setminus A)$. We consider both alternatives.

- Let $t = s^\frown\langle\checkmark\rangle \in \mathsf{divergences}(P \setminus A)$. As from (1) $\mathsf{divergences}(P) = \emptyset$ (i.e., t cannot arise from a divergence of P), by definition there exists $u \in \mathsf{traces}^\omega(P)$ such that $s_1 = u \upharpoonright (\Sigma \backslash A)$ is finite and $t = s_1{}^\frown s_2{}^\frown\langle\checkmark\rangle$. However, by (2), $u \upharpoonright (\Sigma \backslash A)$ cannot be finite for any infinite trace u of P. Due to the contradiction, this case is not possible.
- Therefore, $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}(P \setminus A)$. Therefore, there exists $t' = s'^\frown\langle\checkmark\rangle \in \mathsf{traces}(P)$, such that $s = s' \upharpoonright (\Sigma \backslash A)$. Since $V \in G(P \setminus A)$, by construction there exists $V' \in \mathsf{G}(P)$ with $V' \subseteq V$ and $V' \cap A = \emptyset$. By induction hypothesis for t' and P, s' contains an event from $V' \subseteq V$. But $V' \cap A = \emptyset$. Hence, $s = s' \upharpoonright (\Sigma \backslash A)$ contains an event from V' and therefore from V.

$\square$

- $\mathsf{G}(X) = \emptyset$. $\square$
- $\mathsf{G}(\mu X \bullet P) = \mathsf{G}(P)$.

Proof. Let $V \in \mathsf{G}(\mu X \bullet P)$ and $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(\widehat{\mu X \bullet P})$. We have $\widehat{\mu X \bullet P} = (\mu X \bullet P)(\overline{DIV}) = P^* = \bigcap_{i=0}^{\infty} P^n$, where $P^0 = DIV$, $P^{n+1} = P(P^n, \overline{DIV})$. Since $t \in \mathsf{traces}_\perp(P^*)$, $t \in \mathsf{traces}_\perp(P^n)$ for every $n \in \mathbb{N}$. Therefore, $t = s^\frown\langle\checkmark\rangle \in \mathsf{traces}_\perp(P^1) = \mathsf{traces}_\perp(P(DIV, \overline{DIV})) = \mathsf{traces}_\perp(\widehat{P})$. By construction, $V \in \mathsf{G}(P)$. Therefore, by induction hypothesis for P and t, s contains an event from V. $\square$

$\square$

Proposition 4.6.4. *Let $P(X, Y_1, \ldots, Y_n) = P(X, \overline{Y})$ be a CSP term whose free variables are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $\mathsf{C}_X : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.13. If $(U, V) \in \mathsf{C}_X(P)$, then for all processes $T_1, T_2, \Theta_1, \ldots, \Theta_n \in \mathcal{T}^{\Downarrow}$, $d_V(P(T_1, \overline{\theta}), P(T_2, \overline{\theta})) \leq \frac{1}{2} d_U(T_1, T_2)$.*

Proof. Structural induction on P. Let us take arbitrary $T_1, T_2, \Theta_1, \ldots, \Theta_n \in \mathcal{T}^{\Downarrow}$.

- $\mathsf{C}_X(P) = \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$ whenever X is not free in P.

  Proof. Let $(U, V) \in \mathsf{C}_X(P)$.

  $\frac{1}{2}d_U(T_1, T_2) \geq 0 = d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) = d_V(P(\overline{\Theta}), P(\overline{\Theta}))$. □

- $\mathsf{C}_X(a \longrightarrow P) = \mathsf{C}_X(P) \cup \{(U, V) \in \mathsf{N}_X(P) \mid a \in V\}$.

  Proof. Suppose $(U, V) \in \mathsf{C}_X(a \longrightarrow P)$. By construction, $(U, V) \in \mathsf{C}_X(P)$, or $(U, V) \in \mathsf{N}_X(P)$ and $a \in V$. We consider both cases.

  - Suppose $(U, V) \in \mathsf{C}_X(P)$. Then:
  $$\begin{aligned} \tfrac{1}{2}d_U(T_1, T_2) &\geq d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) && \text{// induction hypothesis} \\ &\geq d_V(a \longrightarrow P(T_1, \overline{\Theta}), a \longrightarrow P(T_2, \overline{\Theta})) && \text{// Lemma 4.5.8} \end{aligned}$$
  - Suppose $(U, V) \in \mathsf{N}_X(P)$ and $a \in V$.
  $$\begin{aligned} \tfrac{1}{2}d_U(T_1, T_2) &\geq \tfrac{1}{2}d_V(P(T_1, \overline{\Theta}), P(T_2, \overline{\Theta})) && \text{// } (U, V) \in \mathsf{N}_X(P), \\ & && \text{// Proposition 4.6.2} \\ &= d_V(a \longrightarrow P(T_1, \overline{\Theta}), a \longrightarrow P(T_2, \overline{\Theta})) && \text{// } a \in V \end{aligned}$$

  □

- $\mathsf{C}_X(P_1 \oplus P_2) = \mathsf{C}_X(P_1) \cap \mathsf{C}_X(P_2) = \{(U_1 \cap U_2, V_1 \cup V_2) \mid (U_i, V_i) \in \mathsf{C}_X(P_i)\}$ for $\oplus \in \{\sqcap, \Box, \underset{A}{\|}\}$.

  Proof. Suppose $(U, V) \in \mathsf{C}_X(P_1 \oplus P_2)$. By construction, there exist $(U_1, V_1) \in \mathsf{C}_X(P_1)$ and $(U_2, V_2) \in \mathsf{C}_X(P_2)$, such that $U = U_1 \cap U_2$ and $V = V_1 \cup V_2$. Therefore, $(U, V) \in \mathsf{C}_X(P_1)$, $(U, V) \in \mathsf{C}_X(P_2)$ (antitoneness). Then:

  $d_V((P_1 \oplus P_2)(T_1, \overline{\Theta}), (P_1 \oplus P_2)(T_2, \overline{\Theta}))$

  $= d_V(P_1(T_1, \overline{\Theta}) \oplus P_2(T_1, \overline{\Theta}), P_1(T_2, \overline{\Theta}) \oplus P_2(T_2, \overline{\Theta}))$

  // ultrametric inequality
  $$\begin{aligned} \leq max\{\ & d_V(P_1(T_1, \overline{\Theta}) \oplus P_2(T_1, \overline{\Theta}), P_1(T_2, \overline{\Theta}) \oplus P_2(T_1, \overline{\Theta})), \\ & d_V(P_1(T_2, \overline{\Theta}) \oplus P_2(T_1, \overline{\Theta}), P_1(T_2, \overline{\Theta}) \oplus P_2(T_2, \overline{\Theta}))\} \end{aligned}$$
  // Lemma 4.5.7
  $$\begin{aligned} \leq max\{\ & d_V(P_1(T_1, \overline{\Theta}), P_1(T_2, \overline{\Theta})) && \text{// } \leq \tfrac{1}{2}d_U(T_1, T_2)\text{, induction hypothesis for } P_1 \\ & d_V(P_2(T_1, \overline{\Theta}), P_2(T_2, \overline{\Theta}))\} && \text{// } \leq \tfrac{1}{2}d_U(T_1, T_2)\text{, induction hypothesis for } P_2 \end{aligned}$$
  $\leq \frac{1}{2}d_U(T_1, T_2)$ □

- $\mathsf{C}_X(P_1 \fatsemi P_2) = \mathsf{C}_X(P_1) \cap (\mathsf{C}_X(P_2) \cup \{(U,V) \in \mathsf{N}_X(P_2) \mid V \in \mathsf{G}(P_1)\}$.

  Proof. Suppose $(U,V) \in \mathsf{C}_X(P_1 \fatsemi P_2)$. By construction, $(U,V) \in \mathsf{C}_X(P_1 \fatsemi P_2)$ yields 2 possibilities:

  - $(U,V) \in \mathsf{C}_X(P_1) \cap \mathsf{C}_X(P_2)$. The proof is the same as the proof for $\sqcap, \Box$ and $\underset{A}{\|}$.
  - $(U,V) \in \mathsf{C}_X(P_1) \cap \{(U,V) \in \mathsf{N}_X(P_2) \mid V \in \mathsf{G}(P_1)\}$. Again, using the ultrametric inequality:

  $$\begin{array}{l} d_V(P_1(T_1,\overline{\Theta}) \fatsemi P_2(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta}) \fatsemi P_2(T_2,\overline{\Theta})) \\ \le max\{\ d_V(P_1(T_1,\overline{\Theta}) \fatsemi P_2(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta}) \fatsemi P_2(T_1,\overline{\Theta})), \\ \qquad\qquad d_V(P_1(T_2,\overline{\Theta}) \fatsemi P_2(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta}) \fatsemi P_2(T_2,\overline{\Theta}))\} \\ \text{// Lemma 4.5.7} \\ \le max\{\ d_V(P_1(T_1,\overline{\Theta}), P_1(T_2,\overline{\Theta})), \ \text{//} \le \frac{1}{2}d_U(T_1,T_2)\text{, induction hypothesis for } P_1 \\ \qquad\qquad d_V(P_2(T_1,\overline{\Theta}), P_2(T_2,\overline{\Theta}))\} \ \text{//} \le \frac{1}{2}d_U(T_1,T_2)\text{, Prop. A.4.1 (1) and 4.5.11} \\ \le \frac{1}{2}d_U(T_1,T_2) \end{array}$$

  □

- $\mathsf{C}_X(P \setminus A) = \{(U,V) \mid (U,V') \in \mathsf{C}_X(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\}$.

  Proof. Suppose $(U,V) \in \mathsf{C}_X(P \setminus A)$. By construction, there exists V' such that $(U,V') \in \mathsf{C}_X(P)$, $V' \subseteq V$ and $V' \cap A = \emptyset$.

  $$\begin{array}{lll} \frac{1}{2}d_U(T_1,T_2) & \ge d_{V'}(P(T_1,\overline{\Theta}), P(T_2,\overline{\Theta})) & \text{// induction hypothesis} \\ & \ge d_{V'}(P(T_1,\overline{\Theta}) \setminus A, P(T_2,\overline{\Theta}) \setminus A) & \text{// } V' \cap A = \emptyset\text{, Lemma 4.5.9} \\ & \ge d_V(P(T_1,\overline{\Theta}) \setminus A, P(T_2,\overline{\Theta}) \setminus A) & \text{// } V' \subseteq V,\ U \mapsto d_U \text{ antitone} \end{array}$$ □

- $\mathsf{C}_X(P[\![R]\!]) = \{(U,V) \mid (U,V') \in \mathsf{C}_X(P) \wedge R(V') \subseteq V\}$.

  Proof. Suppose $(U,V) \in \mathsf{C}_X(P[\![R]\!])$. By construction, there exists V' such that $(U,V') \in \mathsf{C}_X(P)$ and $R(V') \subseteq V$.

  $$\begin{array}{lll} \frac{1}{2}d_U(T_1,T_2) & \ge d_{V'}(P(T_1,\overline{\Theta}), P(T_2,\overline{\Theta})) & \text{// induction hypothesis} \\ & \ge d_{R(V')}(P(T_1,\overline{\Theta})[\![R]\!], P(T_2,\overline{\Theta})[\![R]\!]) & \text{// Lemma 4.5.10} \\ & \ge d_V(P(T_1,\overline{\Theta})[\![R]\!], P(T_2,\overline{\Theta})[\![R]\!]) & \text{// } R(V') \subseteq V,\ U \mapsto d_U \text{ antitone} \end{array}$$ □

- $\mathsf{C}_X(X) = \emptyset$. □

- $\mathsf{C}_X(\mu Y \bullet P) = \{(U,V) \mid (U',V') \in \mathsf{C}_X(P) \wedge (V',V') \in \mathsf{N}_Y(P) \wedge U \subseteq U' \wedge V' \subseteq V\}$ if $Y \neq X$

Proof. Suppose $(U, V) \in \mathsf{C}_X(\mu Y \,.\, P)$ for $X \neq Y$ and X, Y free in $P(X, Y, Z_1, \ldots, Z_n)$.

Then by construction, there exist $U', V' \subseteq \Sigma$ such that:

1. $(U', V') \in \mathsf{C}_X(P)$
2. $U \subseteq U'$, $V' \subseteq V$
3. $(V', V') \in \mathsf{N}_Y(P)$

Since $(U', V') \in \mathsf{C}_X(P)$, by induction hypothesis we have the following for all T_1, T_2, ξ, $\overline{\Theta} \in \mathcal{T}^{\Downarrow}$:

$$\frac{1}{2} d_{U'}(T_1, T_2) \geq d_{V'}(P(T_1, \xi, \overline{\Theta}), P(T_2, \xi, \overline{\Theta})) \tag{A.4}$$

Since $(V', V') \in \mathsf{N}_Y(P)$, from Proposition 4.6.2:

$$d_{V'}(T_1, T_2) \geq d_{V'}(P(\xi, T_1, \overline{\Theta}), P(\xi, T_2, \overline{\Theta})) \tag{A.5}$$

Let $P_1(Y) = P(T_1, Y, \overline{\Theta})$, $P_2(Y) = P(T_2, Y, \overline{\Theta})$. Then, $(\mu Y \,.\, P)(T_1, \overline{\theta}) = P_1^* = \bigcap_{i=0}^{\infty} P_1^n$ and $(\mu Y \,.\, P)(T_2, \overline{\theta}) = P_2^* = \bigcap_{i=0}^{\infty} P_2^n$, where for $i = 1, 2$, $P_i^0 = \bot = DIV$, $P_i^{n+1} = P(T_i, P_i^n, \overline{\theta}) = P_i(P_i^n)$.

We will prove by induction that for every $n \geq 1$:

$$d_{V'}(P_1^n, P_2^n) \leq \frac{1}{2} d_{U'}(T_1, T_2) \tag{A.6}$$

– Let $n = 1$.

$$\begin{aligned} \tfrac{1}{2} d_{U'}(T_1, T_2) &\geq d_{V'}(P(T_1, DIV, \overline{\Theta}), P(T_2, DIV, \overline{\Theta})) \quad // \text{ from (A.4)} \\ &= d_{V'}(P_1^1, P_2^1) \end{aligned}$$

– Suppose $d_{V'}(P_1^n, P_2^n) \leq \frac{1}{2} d_{U'}(T_1, T_2)$.

$$\begin{aligned} d_{V'}(P_1^{n+1}, P_2^{n+1}) &= d_{V'}(P(T_1, P_1^n, \overline{\Theta}), P(T_2, P_2^n, \overline{\Theta})) \\ &\quad // \text{ ultrametric inequality} \\ &\leq max\{d_{V'}(P(T_1, P_1^n, \overline{\Theta}), P(T_2, P_1^n, \overline{\Theta})), \\ &\qquad d_{V'}(P(T_2, P_1^n, \overline{\Theta}), P(T_2, P_2^n, \overline{\Theta}))\} \\ &\leq max\{\tfrac{1}{2} d_{U'}(T_1, T_2), \qquad // \text{ from (A.4)} \\ &\qquad d_{V'}(P_1^n, P_2^n)\} \qquad // \text{ from (A.5)} \\ &\leq max\{\tfrac{1}{2} d_{U'}(T_1, T_2), \\ &\qquad \tfrac{1}{2} d_{U'}(T_1, T_2\} \qquad // \text{ from (A.6), local i.h.} \\ &\leq \tfrac{1}{2} d_{U'}(T_1, T_2) \end{aligned}$$

Now suppose that $d_{V'}(P_1^*, P_2^*) > \frac{1}{2}d_{U'}(T_1, T_2)$ and let $d_{U'}(T_1, T_2) = 2^{-k}$. Let without loss of generality, P_1^* and P_2^* differ on the sets of their divergences and let there exists $s \in \mathsf{divergences}(P_1^*)$ such that $\mathsf{length}_{V'}(s) < k + 1$ and $s \notin \mathsf{divergences}(P_2^*)$. Then, since $P_i^* = \bigcap_{n=0}^{\infty} P_i^n$, there exists $l \in \mathbb{N}$ such that $s \notin P_2^l$, but for all $n \in \mathbb{N}$, $s \in P_1^n$. But then $d_{V'}(P_1^l, P_2^l) > 2^{-(k+1)} = \frac{1}{2}d_{U'}(T_1, T_2)$, which is a contradiction with (A.6). Therefore, $d_{V'}(P_1^*, P_2^*) \leq \frac{1}{2}d_{U'}(T_1, T_2)$. Then, since $U \subseteq U'$ and $V' \subseteq V$, by antitoneness, $d_V((\mu Y \bullet P)(T_1, \overline{\Theta}), (\mu Y \bullet P)(T_2, \overline{\Theta})) = d_V(P_1^*, P_2^*) \leq \frac{1}{2}d_U(T_1, T_2)$. □

□

Proposition 4.6.6. *Let $P(X_1, \ldots, X_n) = P(\overline{X})$ be a CSP term whose free variables are contained within the set $\{X_1, \ldots, X_n\}$. Let $\mathsf{F} : \mathsf{CSP} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ be defined recursively on the structure of P as shown in Figure 4.14. If $(U, V) \in \mathsf{F}(P)$, then, for any collection of U-fair livelock-free processes $\theta_1, \ldots, \theta_n \in \mathcal{T}^{\Downarrow}$, the process $P(\theta_1, \ldots, \theta_n)$ is livelock-free and V-fair.*

Proof. Structural induction on P.

- $\mathsf{F}(\mathit{STOP}) = \mathsf{F}(\mathit{SKIP}) = \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$.

  Proof. *STOP* and *SKIP* are livelock-free and do not contain infinite traces. □

- $\mathsf{F}(a \longrightarrow P) = \mathsf{F}(P)$

  Proof. Let $(U, V) \in \mathsf{F}(a \longrightarrow P)$, $\theta_1, \ldots, \theta_n$ be a collection of livelock-free U-fair processes.

  Since $(U, V) \in \mathsf{F}(a \longrightarrow P)$, by construction we have $(U, V) \in \mathsf{F}(P)$. Therefore by induction hypothesis, $P(\overline{\Theta})$ is livelock-free and V-fair. Therefore, $a \longrightarrow P(\overline{\Theta})$ is livelock-free.

  We will prove that $a \longrightarrow P(\overline{\Theta})$ is V-fair. Let $u \in \mathsf{traces}^{\omega}(a \longrightarrow P(\overline{\Theta}))$. Therefore, by Lemma 4.4.2, there exists $u' \in \mathsf{traces}^{\omega}(P(\overline{\Theta}))$, such that $u = \langle a \rangle ^\frown u'$. Since $P(\overline{\Theta})$ is V-fair, u' contains infinitely many events from V, and so does therefore u. Hence, $a \longrightarrow P(\overline{\Theta})$ is V-fair. □

- $\mathsf{F}(P_1 \oplus P_2) = \mathsf{F}(P_1) \cap \mathsf{F}(P_2)$ for $\{\sqcap, \square\}$

  Proof. Let $(U, V) \in \mathsf{F}(P_1 \oplus P_2)$, $\theta_1, \dots, \theta_n$ be a collection of livelock-free U-fair processes.

  Since $(U, V) \in \mathsf{F}(P_1 \oplus P_2)$, by construction, $(U, V) \in \mathsf{F}(P_1)$ and $(U, V) \in \mathsf{F}(P_2)$. Therefore, by induction hypothesis, $P_1(\overline{\Theta})$ and $P_2(\overline{\Theta})$ are livelock-free and V-fair. Therefore, $P_1(\overline{\Theta}) \oplus P_2(\overline{\Theta})$ is livelock-free.

  Let $u \in \mathsf{traces}^\omega(P_1(\overline{\Theta}) \oplus P_2(\overline{\Theta}))$. Then, by Lemma 4.4.3, $u \in \mathsf{traces}^\omega(P_1(\overline{\Theta}))$ or $u \in \mathsf{traces}^\omega(P_2(\overline{\Theta}))$. Let without loss of generality the former holds. Then, since $P_1(\overline{\Theta})$ is V-fair, u contains infinitely many events from V. Therefore, $P_1(\overline{\Theta}) \oplus P_2(\overline{\Theta})$ is V-fair. □

- $\mathsf{F}(P_1 \mathbin{⨾} P_2) = \mathsf{F}(P_1) \cap \mathsf{F}(P_2)$

  Proof. Let $(U, V) \in \mathsf{F}(P_1 \mathbin{⨾} P_2)$, $\theta_1, \dots, \theta_n$ be a collection of livelock-free U-fair processes.

  Since $(U, V) \in \mathsf{F}(P_1 \mathbin{⨾} P_2)$, by construction we have $(U, V) \in \mathsf{F}(P_1)$ and $(U, V) \in \mathsf{F}(P_2)$. Therefore by induction hypothesis, $P_1(\overline{\Theta})$ and $P_2(\overline{\Theta})$ are livelock-free and V-fair. Therefore $P_1(\overline{\Theta}) \mathbin{⨾} P_2(\overline{\Theta})$ is livelock-free.

  Let $u \in \mathsf{traces}^\omega(P_1(\overline{\Theta}) \mathbin{⨾} P_2(\overline{\Theta}))$. Then by Lemma 4.4.4, $u \in \mathsf{traces}^\omega(P_1(\overline{\Theta}))$, or $u = t_1 {}^\frown u_2$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(P_1(\overline{\Theta})) \cap \Sigma^{*\checkmark}$ and $u_2 \in \mathsf{traces}^\omega(P_2(\overline{\Theta}))$.

  - Suppose $u \in \mathsf{traces}^\omega(P_1(\overline{\Theta}))$. Since $P_1(\overline{\Theta})$ is V-fair, u contains infinitely many events from V.
  - Suppose $u = t_1 {}^\frown u_2$ with $t_1 {}^\frown \langle \checkmark \rangle \in \mathsf{traces}(P_1(\overline{\Theta})) \cap \Sigma^{*\checkmark}$ and $u_2 \in \mathsf{traces}^\omega(P_2(\overline{\Theta}))$. Since $P_2(\overline{\Theta})$ is V-fair, u_2 contains infinitely many events from V and so does therefore u.

  Therefore, $P_1(\overline{\Theta}) \mathbin{⨾} P_2(\overline{\Theta})$ is V-fair. □

- $$\begin{aligned} \mathsf{F}(P_1 \underset{A}{\|} P_2) = {} & (\mathsf{F}(P_1) \cap \mathsf{F}(P_1)) \ \cup \\ & \{(U_1 \cap U_2, V_1) \mid (U_1, V_1) \in \mathsf{F}(P_1) \wedge (U_2, A) \in \mathsf{F}(P_2)\} \ \cup \\ & \{(U_1 \cap U_2, V_2) \mid (U_2, V_2) \in \mathsf{F}(P_2) \wedge (U_1, A) \in \mathsf{F}(P_1)\} \end{aligned}$$

Proof. Let $(U, V) \in \mathsf{F}(P_1 \underset{A}{\parallel} P_2)$, $\theta_1, \ldots, \theta_n$ be a collection of livelock-free U-fair processes.

Since $(U, V) \in \mathsf{F}(P_1 \underset{A}{\parallel} P_2)$, by construction, $\mathsf{F}(P_1) \neq \emptyset$, $\mathsf{F}(P_2) \neq \emptyset$ and, by induction hypothesis, $P_1(\overline{\Theta})$ and $P_2(\overline{\Theta})$ are livelock-free. Therefore, $P_1(\overline{\Theta}) \underset{A}{\parallel} P_2(\overline{\Theta})$ is livelock-free.

Let $u \in \mathsf{traces}^{\omega}(P_1(\overline{\Theta}) \underset{A}{\parallel} P_2(\overline{\Theta}))$. Therefore by Lemma 4.4.7, there exist $u_1 \in \mathsf{traces}^{\infty}(P_1(\overline{\Theta}))$, $u_2 \in \mathsf{traces}^{\infty}(P_2(\overline{\Theta}))$, such that $u \in u_1 \underset{A}{\parallel} u_2$, and $u_1 \in \Sigma^{\omega}$ or $u_2 \in \Sigma^{\omega}$. Let without loss of generality $u_1 \in \Sigma^{\omega}$. By construction, we have three alternatives for (U, V).

- Suppose $(U, V) \in \mathsf{F}(P_1) \cap \mathsf{F}(P_2)$. By induction hypothesis, $P_1(\overline{\Theta})$ is V-fair. Therefore, u_1 contains infinitely many events from V and so does u.
- Suppose (U, V) is $(U_1 \cap U_2, V)$ with $(U_1, V) \in \mathsf{F}(P_1)$, $(U_2, A) \in \mathsf{F}(P_2)$. Then $U = U_1 \cap U_2 \subseteq U_1$, and therefore $\theta_1, \ldots, \theta_n$ are also U_1-fair. Hence, by induction hypothesis, $P_1(\overline{\Theta})$ is V-fair. Then, u_1 contains infinitely many events from V and so does therefore u.
- Suppose (U, V) is $(U_1 \cap U_2, V)$ with $(U_2, V) \in \mathsf{F}(P_2)$, $(U_1, A) \in \mathsf{F}(P_1)$. Since $U = U_1 \cap U_2$, we have $U \subseteq U_1$, $U \subseteq U_2$, and hence $\theta_1, \ldots, \theta_n$ are all both U_1-fair and U_2-fair. By induction hypothesis for P_1, u_1 contains infinitely many events from A. Since u_1 and u_2 synchronise on the events in A, u_2 contains infinitely many events from A. Therefore, $u_2 \in \Sigma^{\omega}$ and by induction hypothesis for P_2, u_2 contains infinitely many events from V. Hence, u contains infinitely many events from V.

Therefore, $P_1(\overline{\Theta}) \underset{A}{\parallel} P_2(\overline{\Theta})$ is V-fair. □

- $\mathsf{F}(P \setminus A) = \{(U, V) \mid (U, V') \in \mathsf{F}(P) \wedge V' \cap A = \emptyset \wedge V' \subseteq V\}$

Proof. Let $(U, V) \in \mathsf{F}(P \setminus A)$ and $\theta_1, \ldots, \theta_n$ be a collection of livelock-free U-fair processes.

Since $(U, V) \in \mathsf{F}(P \setminus A)$, by construction there exists $V' \subseteq V$, such that $V' \cap A = \emptyset$ and $(U, V') \in \mathsf{F}(P)$. Therefore by induction hypothesis, $P(\overline{\Theta})$ is livelock-free and V'-fair.

Suppose $P(\overline{\Theta}) \setminus A$ is not livelock-free. Therefore, since $P(\overline{\Theta})$ is livelock-free, there exists $u \in \mathsf{traces}^{\omega}(P)$ such that $u \upharpoonright (\Sigma \backslash A)$ is finite. Since $P(\overline{\Theta})$ is V'-fair, u contains infinitely many events from V'. Since $V' \cap A = \emptyset$, $u \upharpoonright (\Sigma \backslash A)$ contains infinitely events from V', which is a contradiction with $u \upharpoonright (\Sigma \backslash A)$ being finite. Therefore, $P(\overline{\Theta}) \setminus A$ is livelock-free.

Let $u \in \mathsf{traces}^{\omega}(P(\overline{\Theta}) \setminus A)$. Then, by Lemma 4.4.5, there exists $u' \in \mathsf{traces}^{\omega}(P(\overline{\Theta}))$, such that $u = u' \upharpoonright (\Sigma \backslash A)$. Since $P(\overline{\Theta})$ is V'-fair, u' contains infinitely many events from V'. Since $V' \cap A = \emptyset$, $u = u' \upharpoonright (\Sigma \backslash A)$ contains infinitely many events from $V' \subseteq V$. Therefore, u contains infinitely many events from V. Therefore, $P(\overline{\Theta}) \setminus A$ is V-fair. $\square$

- $\mathsf{F}(P[\![R]\!]) = \{(U, V) \mid (U, V') \in \mathsf{F}(P) \wedge R(V') \subseteq V\}$

Proof. Let $(U, V) \in \mathsf{F}(P[\![R]\!])$ and $\theta_1 \ldots, \theta_n$ be a collection of livelock-free U-fair processes.

Since $(U, V) \in \mathsf{F}(P[\![R]\!])$, by construction there exists V', such that $R(V') \subseteq V$ and $(U, V') \in \mathsf{F}(P)$. Therefore by induction hypothesis, $P(\overline{\Theta})$ is livelock-free and V'-fair. Hence, $P(\overline{\Theta})[\![R]\!]$ is also livelock-free.

Let $u \in \mathsf{traces}^{\omega}(P(\overline{\Theta})[\![R]\!])$. Then, by Lemma 4.4.6, there exists $u' \in \mathsf{traces}^{\omega}(P(\overline{\Theta}))$, such that $u' \; R \; u$. Since $P(\overline{\Theta})$ is V'-fair, u' contains infinitely many events from V'. Since $u' \; R \; u$, u contains infinitely many events from $R(V') \subseteq V$. Therefore, u contains infinitely many events from V. Hence, $P(\overline{\Theta})[\![R]\!]$ is V-fair. $\square$

- $\mathsf{F}(X) = \{(U, V) \mid U \subseteq V\}$

Proof. Let $(U, V) \in \mathsf{F}(X)$, θ be a livelock-free U-fair process. $X(\theta) = \theta$ is then livelock-free, and since $U \subseteq V$, $X(\theta) = \theta$ is V-fair. $\square$

- $\mathsf{F}(\mu X . P) = \begin{cases} \{(U, V) \mid (W, W) \in \mathsf{C}_X(P) \cap \mathsf{F}(P) \wedge U \subseteq W \subseteq V\} & \text{if } \mu X \bullet P \text{ is open} \\ \mathcal{P}(\Sigma) \times \{V \mid (W, W) \in \mathsf{C}_X(P) \cap \mathsf{F}(P) \wedge W \subseteq V\} & \text{otherwise} \end{cases}$

Proof. Let $P(X, Y_1, \ldots, Y_n)$ be a CSP term whose free variable are contained within the set $\{X, Y_1, \ldots, Y_n\}$. Let $(U, V) \in \mathsf{F}(\mu\, X \centerdot P)$ and $(\mu\, X \centerdot P)$ be open. Let $\theta_1, \ldots, \theta_n$ be a collection of livelock-free U-fair processes.

Since $(U, V) \in \mathsf{F}(\mu\, X \centerdot P)$, by construction there exists W, such that $U \subseteq W \subseteq V$ and $(W, W) \in \mathsf{C}_X(P) \cap \mathsf{F}(P)$. Therefore $\theta_1, \ldots, \theta_n$ are W-fair and, by induction hypothesis:

$$P(\xi, \overline{\Theta}) \text{ is livelock-free and } W\text{-fair for any livelock-free } W\text{-fair process } \xi. \qquad \text{(A.7)}$$

Since $(W, W) \in \mathsf{C}_X(P)$, by Proposition A.4.1 (2), $P(X, \overline{\Theta})$ is contractive in X with respect to the metric d_W. Therefore, from Banach's fixed point theorem, $P(X, \overline{\Theta})$ has a unique fixed point $(\mu\, X \centerdot P)(\overline{\theta}) = \bigcup_{n=0}^{\infty} P^n = P^*$, where $P^0 = \top = STOP$, $P^{n+1} = P(P^n, \overline{\Theta})$.

We will prove by induction that for every $n \in \mathbb{N}$, P^n is livelock-free and W-fair.

- Let $n = 0$. The process $STOP$ is livelock-free and does not contain infinite traces.
- Suppose that P^n is livelock-free and W-fair. From (A.7), $P^{n+1} = P(P^n, \overline{\Theta})$ is also livelock-free and W-fair.

Therefore, for every $n \in \mathbb{N}$, P^n is livelock-free. Then, since by Proposition A.2.3 the set of livelock-free processes is closed, $(\mu\, X \centerdot P)(\overline{\theta}) = \bigcup_{n=0}^{\infty} P^n = P^*$ is livelock-free.

Now let $u \in \mathsf{traces}^{\omega}((\mu\, X \centerdot P)(\overline{\Theta}))$. Then, for every finite prefix t of u, $t \in \mathsf{traces}(P^*)$, i.e., there exists some sufficiently large n_t, such that $t \in \mathsf{traces}(P^{n_t})$.

If there exists $m \in \mathbb{N}$ such that for each prefix t of u, $t \in \mathsf{traces}(P^m)$, then $u \in \mathsf{traces}^{\omega}(P^m)$. In this case, since P^m is W-fair, u contains infinitely many events from W. Then, since $W \subseteq V$, u contains infinitely many events from V.

Otherwise, we can conclude the following:

$$\text{for each } m \in \mathbb{N}, \text{ there exists a prefix } t \text{ of } u, \text{ such that } t \notin \mathsf{traces}(P^m). \qquad \text{(A.8)}$$

Let $\varepsilon = 2^{-k}$ for some $k \in \mathbb{N}$. Since the sequence $\langle P^i \mid i \in \mathbb{N} \rangle$ converges to P^* with respect to the metric d_W, there exists $n_\varepsilon \in \mathbb{N}$, such that for each $n \geq n_\varepsilon$, $d_W(P^*, P^n) < \varepsilon$. From our assumption (A.8) for $m = n_\varepsilon$, there exists t_ε, such that

t_ε is a prefix of u and $t_\varepsilon \notin \mathsf{traces}(P^{n_\varepsilon})$. Then, since $d_W(P^*, P^{n_\varepsilon}) < \varepsilon = 2^{-k}$ and $t_\varepsilon \in \mathsf{traces}(P^*)$, we obtain $\mathsf{length}_W t_\varepsilon \geq k$. Since k was arbitrary, we can conclude that u contains infinitely many events from W. Then again, since $W \subseteq V$, u contains infinitely many events from V. Therefore, $(\mu\, X \centerdot P)(\overline{\theta})$ is V-fair. □

□

A.5 Proofs for Section 4.7

Proposition 4.7.2. *Let P be a structurally finite state process. Let $\Phi : \overline{\mathsf{SFS}} \longrightarrow \mathcal{P}(\mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma))$ and $\delta : \overline{\mathsf{SFS}} \longrightarrow \{\text{true}, \text{false}\}$ be defined recursively on the structure of P as shown in Figures 4.18 and 4.19, respectively. Then, if $\delta(P) = \text{false}$, P is livelock-free. Moreover, if $\Phi(P) = \{(F_1, C_1), \ldots, (F_k, C_k)\}$, then, for each infinite trace u of P, there exists $1 \leq i \leq k$, such that u is fair in F_i and u is co-fair in C_i.*

Proof. Induction on the structure of the $\overline{\mathsf{SFS}}$ process P.

Note that by construction, all fair/co-fair pairs of sets thus generated remain disjoint, i.e., for each $(F, C) \in \Phi(P)$, $F \cap C = \emptyset$. This is key in the rule for parallel composition, where the fair/co-fair data of individual subcomponents enables one to rule out certain pairs for the resulting parallel process. We prove this property only for the case of renaming as for all other cases it follows trivially from the induction hypothesis and the specific construction.

Let us also remark that it might be the case that $\delta(P) = \text{false}$ and $\Phi(P) = \emptyset$, and this indicates that P is livelock free but exhibits only finite traces. We note, however, that if $\delta(P) = \text{false}$ and $\Phi(P) \neq \emptyset$, then for every $(F, C) \in \Phi(P)$, $F \neq \emptyset$. This is true for atomic $\overline{\mathsf{SFS}}$ processes by construction and follows for compound $\overline{\mathsf{SFS}}$ processes by induction hypothesis and construction. We prove the property only for the cases of hiding and renaming where the argument is more subtle.

- For P an atomic $\overline{\mathsf{SFS}}$ process, $\Phi(P)$ and $\delta(P)$ are computed directly from the labelled transition system associated with P (see Section 4.7.2).

- $\delta(a \longrightarrow P) = \delta(P)$ and $\Phi(a \longrightarrow P) = \Phi(P)$.

  Proof. Let $\delta(a \longrightarrow P) = \text{false}$. By construction, $\delta(P) = \text{false}$ and, therefore, by induction hypothesis, P is livelock-free. Hence, by definition, $a \longrightarrow P$ is also livelock-free.

  Let $u \in \mathsf{traces}^\omega(a \longrightarrow P)$. Then, by Lemma 4.4.2, there exists $u' \in \mathsf{traces}^\omega(P)$, such that $u = \langle a \rangle ^\frown u'$. By induction hypothesis for P, there exists $(F, C) \in \Phi(P)$, such that u' is fair in F and co-fair in C. But then u is also fair in F and co-fair in C and, by construction, $(F, C) \in \Phi(a \longrightarrow P)$. $\square$

- $\delta(P_1 \oplus P_2) = \delta(P_1) \vee \delta(P_2)$ and $\Phi(P_1 \oplus P_2) = \Phi(P_1) \cup \Phi(P_2)$ if $\oplus \in \{\sqcap, \Box\}$.

  Proof. Let $\delta(P_1 \oplus P_2) = \text{false}$. By construction, $\delta(P_1) = \text{false}$ and $\delta(P_2) = \text{false}$. Therefore, by induction hypothesis, P_1 and P_2 are livelock-free. Hence, by definition, $P_1 \oplus P_2$ is livelock-free.

  Let $u \in \mathsf{traces}^\omega(P_1 \oplus P_2)$. By Lemma 4.4.3, $u \in \mathsf{traces}^\omega(P_1)$ or $u \in \mathsf{traces}^\omega(P_2)$. Let without loss of generality the former holds. Then, by induction hypothesis for P_1, there exists $(F, C) \in \Phi(P_1)$, such that u is fair in F and co-fair in C. By construction, $\Phi(P_1) \subseteq \Phi(P_1 \oplus P_2)$ and, therefore, $(F, C) \in \Phi(P_1 \oplus P_2)$. $\Box$

- $\delta(P_1 \,⨾\, P_2) = \delta(P_1) \vee \delta(P_2)$ and $\Phi(P_1 \,⨾\, P_2) = \Phi(P_1) \cup \Phi(P_2)$.

  Proof. Let $\delta(P_1 \,⨾\, P_2) = \text{false}$. By construction, $\delta(P_1) = \text{false}$ and $\delta(P_2) = \text{false}$. Therefore, by induction hypothesis, P_1 and P_2 are livelock-free. Hence, by definition, $P_1 \,⨾\, P_2$ is livelock-free.

  Let $u \in \mathsf{traces}^\omega(P_1 \,⨾\, P_2)$. By Lemma 4.4.4, $u \in \mathsf{traces}^\omega(P_1)$, or $u = t^\frown u'$ with $t^\frown\langle\checkmark\rangle \in \mathsf{traces}(P_1) \cap \Sigma^{*\checkmark}$ and $u' \in \mathsf{traces}^\omega(P_2)$. We consider both alternatives.

  – If $u \in \mathsf{traces}^\omega(P_1)$, by induction hypothesis for P_1, there exists $(F, C) \in \Phi(P_1)$, such that u is fair in F and co-fair in C. By construction, $\Phi(P_1) \subseteq \Phi(P_1 \oplus P_2)$ and, therefore, $(F, C) \in \Phi(P_1 \oplus P_2)$.

  – Let $u = t^\frown u'$ where $t^\frown\langle\checkmark\rangle \in \mathsf{traces}(P_1) \cap \Sigma^{*\checkmark}$ and $u' \in \mathsf{traces}^\omega(P_2)$. By induction hypothesis for P_2, there exists $(F, C) \in \Phi(P_2)$, such that u' is fair in F and co-fair in C. The finite prefix t of u does not affect fairness and co-fairness. Therefore, $u = t^\frown u'$ is fair in F and co-fair in C and $(F, C) \in \Phi(P_1 \oplus P_2)$ by construction.

  $\Box$

- $\delta(P \setminus A) = \begin{cases} \text{false} & \text{if } \delta(P) = \text{false and, for each } (F, C) \in \Phi(P),\ F - A \neq \emptyset \\ \text{true} & \text{otherwise} \end{cases}$
  and $\Phi(P \setminus A) = \{(F - A, C \cup A) \mid (F, C) \in \Phi(P)\}$.

Proof. Let $\delta(P \setminus A) = \text{false}$. By construction, $\delta(P) = \text{false}$ and for each $(F, C) \in \Phi(P)$, $F - A \neq \emptyset$. Since $\delta(P) = \text{false}$, by induction hypothesis, P is livelock-free. Suppose $P \setminus A$ can diverge. Since P is livelock-free, by definition, the only alternative is that there exists $u \in \mathsf{traces}^{\omega}(P)$, such that $u \upharpoonright (\Sigma \setminus A)$ is finite. By induction hypothesis for P, there exists $(F, C) \in \Phi(P)$, such that u is fair in F and co-fair in C. By construction, since $\delta(P \setminus A) = \text{false}$, $F - A \neq \emptyset$. Therefore, there exists $b \in F$ such that $b \notin A$ and b occurs infinitely many times in u. But then b should also occur infinitely many times in $u \upharpoonright (\Sigma \setminus A)$, which is a contradiction with $u \upharpoonright (\Sigma \setminus A)$ being finite. Therefore, $P \setminus A$ is livelock-free.

Now, let $u \in \mathsf{traces}^{\omega}(P \setminus A)$. Since $P \setminus A$ is livelock-free, by Lemma 4.4.5, there exists $v \in \mathsf{traces}^{\omega}(P)$ such that $u = v \upharpoonright (\Sigma \setminus A)$. By induction hypothesis for P, there exists $(F, C) \in \Phi(P)$ such that v is fair in F and co-fair in C. Then, since u is obtained by deleting all A-events from v, u is fair in $F - A$ and co-fair in $C \cup A$. Both $F - A \neq \emptyset$ and $(F - A, C \cup A) \in \Phi(P \setminus A)$ are guaranteed by construction.

Let $\delta(P \setminus A) = \text{false}$ and let $(F, C) \in \Phi(P \setminus A)$. We now prove that $F \neq \emptyset$. Since $\delta(P \setminus A) = \text{false}$, by construction we have the following:

$$\text{for each } (F', C') \in \Phi(P), F' - A \neq \emptyset. \tag{A.9}$$

As $(F, C) \in \Phi(P \setminus A)$, by construction $F = F' - A$ for some F' with $(F', C') \in \Phi(P)$. By (A.9), $F' - A \neq \emptyset$, and hence $F \neq \emptyset$. $\square$

- $\delta(P[\![R]\!]) = \delta(P)$ and
  $$\begin{aligned}\Phi(P[\![R]\!]) = \{(F, C) \mid\ & (F', C') \in \Phi(P) \wedge F' \subseteq R^{-1}(F) \wedge F \subseteq R(F') \wedge \\ & C = \{b \in \Sigma \mid R^{-1}(b) \subseteq C'\}\}\end{aligned}$$

Proof. In the proof we use the following notation. For any $A \subseteq \Sigma$, $a, b \in \Sigma$, we write $R(A) = \{b \mid \exists\, a \in A \bullet a\ R\ b\}$ and $R^{-1}(b) = \{a \mid a\ R\ b\}$. Let us also clarify that in the setting of CSP [Ros98] renaming relations are assumed to be total. If an event $a \in \Sigma$ is not renamed to any other event $b \in \Sigma$, it is assumed that a is renamed to itself, and hence $R(\{a\}) \neq \emptyset$.

Let $\delta(P[\![R]\!]) = \text{false}$. By construction, $\delta(P) = \text{false}$. Then by induction hypothesis, P is livelock-free, and hence by definition so is $P[\![R]\!]$.

Let $u \in \mathsf{traces}^\omega(P[\![R]\!])$. By Lemma 4.4.6, there exists $v \in \mathsf{traces}^\omega(P)$, such that $v\ R\ u$, i.e., for every $i \in \mathbb{N}$, $v(i)\ R\ u(i)$. By induction hypothesis for P, there exists $(U', V') \in \Phi(P)$, such that v is fair in F' and co-fair in C'.

Let $C = \{b \in \Sigma \mid R^{-1}(b) \subseteq C'\}$ and let $b \in C$. By construction, for each $a \in R^{-1}(b)$, $a \in C'$ and, therefore, v is co-fair in a. Now suppose for the sake of contradiction that u contains infinitely many occurrences of b. By definition, there exists $a \in \Sigma$, such that $a\ R\ b$ and a occurs infinitely many times in v. Therefore, $a \notin C'$ and $R^{-1}(b) \nsubseteq C'$, which is a contradiction with $R^{-1}(b) \subseteq C'$. Therefore, u contains only finitely many b's and, more generally, u is co-fair in C.

We will construct F from F' in a way that $F \subseteq R(F')$ (which will bound F from above), $F' \subseteq R^{-1}(F)$ (which will bound F from below and will guarantee $F \neq \emptyset$) and u is fair in F. Then $(F, C) \in \Phi(P[\![R]\!])$ by construction.

By induction hypothesis, $F' \neq \emptyset$. Let $F' = \{a_1, \dots, a_m\}$ and for each $1 \le i \le m$, $R(\{a_i\}) = \{b_{i_1}, \dots, b_{i_{n_i}}\}$. As each a_i occurs infinitely many times in v and $v\ R\ u$, for each $1 \le i \le m$, there exists b_{j_i}, such that $a_i\ R\ b_{j_i}$ and b_{j_i} occurs infinitely many times in u. We define $F = \{b_{j_1}, \dots, b_{j_m}\}$. Since $F' \neq \emptyset$, $F \neq \emptyset$. By the construction of F, u is fair in F and $F \subseteq R(F')$. As by construction for each $a_i \in F'$ there exists $b_{j_i} \in F$ with $a_i\ R\ b_{j_i}$, then for every $1 \le i \le m$, $a_i \in R^{-1}(b_{j_i})$. Therefore, $F' \subseteq R^{-1}(F)$.

We will also prove that for any F that satisfies $F \subseteq R(F')$ and $F' \subseteq R^{-1}(F)$, the sets F and C are disjoint. Suppose there exists $b \in \Sigma$ such that $b \in F \cap C$. As $b \in C$, by construction, for each a with $a\ R\ b$, we have $a \in C'$. Since $b \in F$ and by construction $F \subseteq R(F')$, there exists $a \in F'$, such that $a\ R\ b$. Therefore, $a \in C' \cap F'$. This is a contradiction with the induction hypothesis according to which F' and C' are disjoint. Therefore, $F \cap C = \emptyset$. □

- $\delta(P_1 \underset{A}{\|} P_2) = \delta(P_1) \vee \delta(P_2)$ and

  $\Phi(P_1 \underset{A}{\|} P_2) = \{(F, C) \mid F \cap C = \emptyset \wedge (F_i, C_i) \in \Phi(P_i) \text{ for } i = 1, 2 \wedge F = F_1 \cup F_2 \wedge$
  $C = (C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A))\} \cup$

$$\{(F, C) \mid (F, C) \in \Phi(P_1) \wedge F \cap A = \emptyset\} \cup$$
$$\{(F, C) \mid (F, C) \in \Phi(P_2) \wedge F \cap A = \emptyset\}$$

Proof. Let $\delta(P_1 \underset{A}{\parallel} P_2) = \text{false}$. By construction, $\delta(P_1) = \text{false}$ and $\delta(P_2) = \text{false}$. Therefore, by induction hypothesis, P_1 and P_2 are livelock-free. Hence by definition, $P_1 \underset{A}{\parallel} P_2$ is livelock-free.

Let $u \in \mathsf{traces}^\omega(P_1 \underset{A}{\parallel} P_2)$. From Lemma 4.4.7, there exist $u_1 \in \mathsf{traces}^\infty(P_1)$ and $u_2 \in \mathsf{traces}^\infty(P_2)$, such that $u \in u_1 \underset{A}{\parallel} u_2$, and $u_1 \in \Sigma^\omega$ or $u_2 \in \Sigma^\omega$. We will consider three different cases.

- Let $u_1 \in \Sigma^\omega$ and $u_2 \in \Sigma^{*\checkmark}$. By induction hypothesis for P_1, there exists $(F, C) \in \Phi(P_1)$ such that u_1 is fair in F and co-fair in C. Suppose for the sake of the argument that $F \cap A \neq \emptyset$. Then, u_1 contains infinitely many occurrences of events from A. Since P_1 and P_2 synchronise on the events in A, u_2 must also contain infinitely many events from A, which is a contradiction with $u_2 \in \Sigma^{*\checkmark}$. Therefore, $F \cap A = \emptyset$ and, by construction, $(F, C) \in \Phi(P_1 \underset{A}{\parallel} P_2)$. Now, since u_2 is finite and does not affect fairness and co-fairness, u is fair in F and co-fair in C.
- The case where $u_2 \in \Sigma^\omega$ and $u_1 \in \Sigma^{*\checkmark}$ is handled in the same way.
- Let $u_1 \in \Sigma^\omega$ and $u_2 \in \Sigma^\omega$. By induction hypothesis for P_1 and P_2, there exist $(F_1, C_1) \in \Phi(P_1)$ and $(F_2, C_2) \in \Phi(P_2)$, such that u_1 is fair in F_1 and co-fair in C_1 and u_2 is fair in F_2 and co-fair in C_2. We note, that for each $a \in A$, the number of occurrences of a in u_1, u_2 and u is the same due to P_1 and P_2 synchronising on a. Therefore, for each $a \in A$, u_1 contains infinitely many occurrences of a if and only if u_2 contains infinitely many occurrences of a. Hence, $F_1 \cap C_2 \cap A = \emptyset$ and $F_2 \cap C_1 \cap A = \emptyset$.

  Let $F = F_1 \cup F_2$ and $C = (C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A))$.

  We will first prove that $F \cap C = \emptyset$. Suppose for the sake of contradiction that there exists $b \in \Sigma$ such that $b \in F \cap C$. Since $b \in F$, by construction, $b \in F_1$ or $b \in F_2$. Let without loss of generality $b \in F_1$. We will consider the cases $b \in A$ and $b \notin A$.

- Suppose $b \in A$. Since $b \in F_1$, u_1 is fair in b and, therefore, $b \notin C_1$. Since $b \in C$ and $b \in A$, $b \in C_1 \cap A$ or $b \in C_2 \cap A$. As $b \notin C_1$, $b \in C_2 \cap A$. Therefore, $b \in F_1 \cap C_2$ which is a contradiction with $F_1 \cap C_2 \cap A = \emptyset$. Therefore, this case is not possible.
- Suppose $b \notin A$. Since $b \in C$, $b \in C_1$ and $b \in C_2$. Therefore, $b \in F_1 \cap C_1$ which is a contradiction with the induction hypothesis by which F_1 and C_1 are disjoint. Therefore, this case is not possible either.

Therefore, $F \cap C = \emptyset$.

Now, for any event $b \in \Sigma$, if $b \in F_1$ or $b \in F_2$, i.e., b has infinitely many occurrences in u_1 or u_2, then b has infinitely many occurrences in $u \in u_1 \underset{A}{\parallel} u_2$ as well. Therefore, u is fair in $F_1 \cup F_2$.

Let for some $a \in A$, $a \in C_1$ or $a \in C_2$ and let without loss of generality the former holds. Then, a occurs only finitely many times in u_1, and since P_1 and P_2 synchronise on a, a occurs only finitely many times in u_2 and u as well. Therefore, u is co-fair in a and, more generally, in $(C_1 \cap A) \cup (C_2 \cap A)$. Now let $b \in (C_1 - A) \cap (C_2 - A)$. Therefore, $b \notin A$, $b \in C_1$ and $b \in C_2$. Therefore, since b occurs only finitely often in both u_1 and u_2, b occurs only finitely often in u as well. Therefore, u is also co-fair in $(C_1 - A) \cap (C_2 - A)$. Hence, u is co-fair in $(C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A))$ and $(F_1 \cup F_2, (C_1 \cap A) \cup (C_2 \cap A) \cup ((C_1 - A) \cap (C_2 - A)) \in \Phi(P_1 \underset{A}{\parallel} P_2)$ by construction.

□

□

Bibliography

[Ace03] L. Aceto. Some of my favourite results in classic process algebra. *Bulletin of the EATCS*, 81:90–108, 2003.

[ADK$^+$05] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME'05)*, volume 3725 of *LNCS*, pages 254–268. Springer, 2005.

[AGL$^+$12] P. Armstrong, M. Goldsmith, G. Lowe, J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. Recent developments in FDR. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *LNCS*, pages 699–704. Springer, 2012.

[AJS05] A. E. Abdallah, C. B. Jones, and J. W. Sanders. *Communicating Sequential Processes: the first 25 years*, volume 3525 of *LNCS*. Springer, 2005.

[ASST10] I. Abdelhalim, J. Sharp, S. A. Schneider, and H. Treharne. Formal verification of Tokeneer behaviours modelled in fUML using CSP. In *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *LNCS*, pages 371–387. Springer, 2010.

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335:131–146, 2005.

[BAS02] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[BC05] G. H. Broadfoot and Verum Consultants. Introducing formal methods into industry using Cleanroom and CSP. *Dedicated Systems Magazine, Q*, 1:2005, 2005.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[BCD$^+$05] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BG02] M. Bravetti and R. Gorrieri. Deciding and axiomatizing weak ST bisimulation for a process algebra with recursion and action refinement. *ACM Transactions on Computational Logic*, 3(4):465–520, 2002.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BHvMW09] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bie08] A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKWW08] A. Biere, D. Kroening, G. Weissenbacher, and C. M. Wintersteiger. *Digitaltechnik*. Springer, 2008.

[BLR11] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[BM08] A. R. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Aspects of Computing*, 20(4-5):379–405, 2008.

[BMWK10] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Context-aware counter abstraction. *Formal Methods in System Design*, 36(3):223–245, 2010.

[BPG08] M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*, volume 5123 of *LNCS*, pages 135–148. Springer, 2008.

[BR01] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[Bra11] A. R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[Bra12] A. R. Bradley. Understanding IC3. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *LNCS*, pages 1–14. Springer, 2012.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[BZ82] J. W. De Bakker and J. I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[CC02] P. Cousot and R. Cousot. On abstraction in software verification. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 37–56. Springer, 2002.

[CCG$^+$02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An open-source tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[CCGP] S. Chaki, E. M. Clarke, D. Giannakopoulou, and C. S. Pasareanu. Abstraction and assume-guarantee reasoning for automated software verification. Technical report, 05.02, Research Institute for Advanced Computer Science (RIACS).

[CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.

[CCO$^+$05] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[CCOS04] S. Chaki, E. M. Clarke, J. Ouaknine, and N. Sharygina. Automated, compositional and iterative deadlock detection. In *Proceedings of the 2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE'04)*, pages 201–210. IEEE, 2004.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1981.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'03)*, volume 697 of *LNCS*, pages 450–462. Springer, 1993.

[CGJ$^+$00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[CGJ$^+$03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[CKOS04] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 85–96. Springer, 2004.

[CKOS05] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Computational challenges in bounded model checking. *STTT*, 7(2):174–183, 2005.

[CKS05] B. Cook, D. Kroening, and N. Sharygina. Symbolic model checking for asynchronous Boolean programs. In *Proceedings of the 12th International SPIN Workshop (SPIN'05)*, volume 3639 of *LNCS*, pages 75–90. Springer, 2005.

[CKSY05] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[CKY03] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC'03)*, pages 368–371. ACM, 2003.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC'71)*, pages 151–158. ACM, 1971.

[Cou05] P. Cousot. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`, February–May 2005.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. *Electronic Notes in Theoretical Computer Science*, 89(3):417–432, 2003.

[Cra57] W. Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[CY10] A. Chawdhary and H. Yang. Metric spaces and termination analyses. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10)*, volume 6461 of *LNCS*, pages 156–171. Springer, 2010.

[DHKR11] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *Proceedings of the 18th International Symposium on Static Analysis (SAS'11)*, volume 6887 of *LNSC*, pages 351–368. Springer, 2011.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dim10] A. Dimovski. A compositional method for deciding program termination. In *ICT Innovations*, volume 83, pages 71–80. Springer CCIS, 2010.

[DKKW11] A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 356–371. Springer, 2011.

[DKPW10] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'10)*, volume 5944 of *LNCS*, pages 129–145. Springer, 2010.

[DKR10] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *LNCS*, pages 280–295. Springer, 2010.

[DKR11a] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[DKR11b] A. F. Donaldson, D. Kroening, and P. Rümmer. SCRATCH: a tool for automatic analysis of DMA races. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'11)*, pages 311–312. ACM, 2011.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[EB05] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[EMB11] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'11)*, pages 125–134. IEEE, FMCAD Inc., 2011.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 463–478. Springer, 1993.

[ES03a] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[ES03b] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Compututer Science*, 89(4):543–560, 2003.

[G$^+$05] M. Goldsmith et al. *Failures-Divergence Refinement. FDR2 User Manual.* Formal Systems (Europe) Ltd., June 2005.

[Gan80] R. O. Gandy. An early proof of normalization by A.M. Turing. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, volume 267, pages 453–455. Academic Press, 1980.

[GL91] O. Grumberg and D. E. Long. Model checking and modular verification. In *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91)*, volume 527 of *LNCS*, pages 250–265. Springer, 1991.

[GLST05] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL'05)*, volume 40, pages 122–131. ACM, 2005.

[GLT88] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Science 7. Cambridge University Press, 1988.

[God95] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. PhD thesis, Universite de Liege, 1995.

[God99] P. Godefroid. Exploiting symmetry when model-checking software. In *Proceedings of Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII)*, pages 257–275, 1999.

[Gol04] M. Goldsmith. Operational semantics for fun and profit. In *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 265–274. Springer, 2004.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[GW05] M. Goldsmith and P. Whittaker. A CSP frontend for probabilistic tools. Technical report, Formal Systems (Europe Ltd), 2005. `http://www.forward-project.org.uk/PDF_Files/D14.pdf`.

[HJMQ03] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 262–274. Springer, 2003.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 58–70. ACM, 2002.

[HKNP06] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*. Springer, 2006.

[HM80] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming (ICALP'80)*, volume 85 of *LNCS*, pages 299–309. Springer, 1980.

[Hoa80] C. A. R. Hoare. A model for communicating sequential processes. In *On the Construction of Programs*, pages 229–254. 1980.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[HQR00] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'00)*, pages 245–253. IEEE, 2000.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[Kah12] V. Kahlon. Schedule insensitivity reduction. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12)*, volume 7212 of *LNCS*, pages 99–114. Springer, 2012.

[KB05] R. H. Katz and G. Borriello. *Contemporary Logic Design, Second Edition.* Prentice Hall, 2005.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KS83] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 228–240. ACM, 1983.

[KS08] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View.* Springer, 2008.

[Kur94] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton University Press, 1994.

[KWG09] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.

[LcW06] S. Leue, A. Ştefănescu, and W. Wei. A livelock freedom analysis for infinite state asynchronous reactive systems. In *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR'06)*, volume 4137 of *LNCS*, pages 79–94. Springer, 2006.

[LcW08] S. Leue, A. Ştefănescu, and W. Wei. Dependency analysis for control flow cycles in reactive communicating processes. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software (SPIN'08)*, volume 5156 of *LNCS*. Springer, 2008.

[Low93] G. Lowe. *Probabilities and Priorities in Timed CSP*. PhD thesis, Oxford University Computing Laboratory, 1993.

[Low95] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[Low98] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[Low04] G. Lowe. On the application of counterexample-guided abstraction refinement and data independence to the parameterised model checking problem. In *Proceedings of the 3rd International Workshop on Automatic Verification of Infinite-State Systems (AVIS'04)*, 2004.

[Mar95] J. M. R. Martin. Deadlock checker user guide and technical manual, 1995.

[Mar96] J. M. R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

[McM03] K. L. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[McM06] K. L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, 1999.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[MJ97] J. M. R. Martin and S. A. Jassim. How to design deadlock-free networks using CSP and verification tools - a tutorial introduction, 1997. `http://wotug.org/parallel/theory/formal/csp/Deadlock/#overview`.

[MMZ$^+$01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*, pages 530–535. ACM, 2001.

[MPW93] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[OPRW11] J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. Static livelock analysis in CSP. In *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR'11)*, volume 6901 of *LNCS*, pages 389–403. Springer, 2011. Winner of the CONCUR 2011 Best Paper Award.

[OPRW13] J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. A static analysis framework for livelock freedom in CSP. In Joost-Pieter Katoen and Barbara König, editors, *Logical Methods in Computer Science, Special Issue: Selected Papers of the 22nd Conference on Concurrency Theory (CONCUR'11).* 2013. To appear.

[Oua00] J. Ouaknine. *Discrete Analysis of Continuous Behaviour in Real-Time Concurrent Systems.* PhD thesis, University of Oxford, 2000.

[Pap04] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley, 2004.

[Par81] D. M. R. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer, 1981.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.

[Pel98] D. Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.

[Pel08] R. Pelánek. Fighting state space explosion: Review and evaluation. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *LNCS*, pages 37–52. Springer, 2008.

[PG86] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. *DAIMI FN-19, Department of Computer Science, Aarhus University*, 1981.

[Plo04] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[POR09] H. Palikareva, J. Ouaknine, and A. W. Roscoe. Faster FDR counterexample generation using SAT-solving. In Markus Roggenbach, editor, *Proceedings of the 9th International Workshop on Automated Verification of Critical Systems (AVoCS'09)*, volume 23 of *Electronic Communications of the EASST*, 2009.

[POR12] H. Palikareva, J. Ouaknine, and A. W. Roscoe. SAT-solving in CSP trace refinement. *Science of Computer Programming, special issue on Automated Verification of Critical Systems*, 77(10-11):1178–1197, 2012.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PY96] A. Parashkevov and J. Yantchev. ARC – a tool for efficient refinement and equivalence checking for CSP. In *Proceedings of the 2nd IEEE International Conference on Algorithm and Architectures for Parallel Processing*, pages 68–75. IEEE, 1996.

[qin] QinetiQ. `http://www.qinetiq.com/`.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, Springer, 1982.

[QW04] S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*, pages 14–24. ACM, 2004.

[RB85] A. W. Roscoe and S. D. Brookes. Deadlock analysis in networks of communicating processes. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*, pages 305–324. Springer, 1985.

[RD87] A. W. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.

[RG05] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.

[RGG$^+$95] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP, or How to check 10^{20} dining philosophers for deadlock. In *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *LNCS*, pages 133–152. Springer, 1995.

[RGM$^+$03] A. W. Roscoe, M. H. Goldsmith, N. Moffat, T. Whitworth, and I. Zakiuddin. Watchdog transformations for property-oriented model checking. In *Proceedings of the International Symposium of Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 600–616. Springer, 2003.

[Ros82] A. W. Roscoe. *A Mathematical Theory of Communicating Processes.* PhD thesis, Oxford University, 1982.

[Ros90] A. W. Roscoe. Maintaining consistency in distributed databases. Technical Report PRG-87, Oxford University Computing Laboratory, 1990.

[Ros94] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: essays in Honour of C.A.R. Hoare*, chapter 21. Prentice-Hall, 1994.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Ros10] A. W. Roscoe. CSP is expressive enough for π. 2010. Available at `http://www.cs.ox.ac.uk/files/2340/hoarecomplete2.pdf`.

[Ros11a] A. W. Roscoe. On the expressiveness of CSP. To appear. Available at `http://www.cs.ox.ac.uk/files/1383/expressive.pdf`, February 2011.

[Ros11b] A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2011. `http://www.cs.ox.ac.uk/ucs/`.

[RR86] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In *Proceedings of the 13th International Colloquium on Automata, Languages and Programming (ICALP'86)*, volume 226 of *LNCS*, pages 314–323. Springer, 1986.

[RRS$^+$01] A. W. Roscoe, P. Ryan, S. Schneider, M. Goldsmith, and G. Lowe. *The Modelling and Analysis of Security Protocols.* Addison-Wesley, 2001.

[RW06] A. W. Roscoe and Z. Wu. Verifying statemate statecharts using CSP and FDR. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM'06)*, volume 4260 of *LNCS*, pages 324–341. Springer, 2006.

[San02] D. Sangiorgi. Types, or: Where's the difference between CCS and π? In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR'02)*, volume 2421 of *LNCS*, pages 76–97. Springer, 2002.

[Sch00] S. Schneider. *Concurrent and Real Time Systems: the CSP Approach.* Wiley Chichester, UK, 2000.

[Sht00] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 480–494. Springer, 2000.

[SLD08] J. Sun, Y. Liu, and J. S. Dong. Model checking CSP revisited: Introducing a process analysis toolkit. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.

[SLDS08] J. Sun, Y. Liu, J. S. Dong, and J. Sun. Bounded model checking of compositional processes. In *Proceedings of the 2nd IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE'08)*, pages 23–30. IEEE, 2008.

[SS71] D. S. Scott and C. Strachey. *Toward a mathematical semantics for computer languages.* Oxford University Computing Laboratory, Programming Research Group, 1971.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[ST05] S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17(4):390–422, 2005.

[STW10] S. Schneider, H. Treharne, and H. Wehrheim. A CSP approach to control in Event-B. In *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM'10)*, volume 6396 of *LNCS*. Springer, 2010.

[STW11] S. Schneider, H. Treharne, and H. Wehrheim. A CSP account of Event-B refinement. In *Proceedings of the 15th International Refinement Workshop (Refine'11)*, volume 55 of *EPTCS*, pages 139–154, 2011.

[Sut75] W. A. Sutherland. *Introduction to Metric and Topological Spaces.* Oxford University Press, 1975.

[Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125, 1968.

[Val97] A. Valmari. Stubborn set methods for process algebras. In *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification (POMIV'96)*, pages 213–231. AMS Press, Inc., 1997.

[Val09] A. Valmari. Bisimilarity minimization in $o(m\ \log n)$ time. In *Proceedings of the 30th International Conference on Applications and Theory of Petri Nets (PETRI NETS'09)*, volume 5606 of *LNCS*, pages 123–142. Springer, 2009.

[ver] Verum BV. `http://www.verum.com/`.

[VF10] A. Valmari and G. Franceschinis. Simple $o(m\ \log n)$ time Markov chain lumping. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *LNCS*, pages 38–52. Springer, 2010.

[vG90] R. J. van Glabbeek. The linear time-branching time spectrum II. In *Proceedings of Theories of Concurrency : Unification and Extension*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.

[vG00] R. J. van Glabbeek. The linear time-branching time spectrum I - the semantics of concrete, sequential processes. In *Handbook of Process Algebra, chapter 1*, pages 3–99. Elsevier, 2000.

[VST09] B. Vajar, S. Schneider, and H. Treharne. Mobile CSP $\|$ B. *ECEASST*, 23, 2009.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

[WK07] X. Wang and M. Z. Kwiatkowska. Compositional state space reduction using untangled actions. *Electronic Notes in Theoretical Computer Science*, 175(3):27–46, 2007.

[YBH01] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the π-Calculus. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 311–322. IEEE Computer Society Press, 2001.

[YJ89] J. Yantchev and C. R. Jesshope. Adaptive, low latency, deadlock-free packet routing for networks of processors. In *IEEE Proceedings of Computers and Digital Techniques*, 1989.