BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

MINIMIZATION OF COUNTING AUTOMATA

MINIMALIZACE AUTOMATŮ S JEDNODUCHÝMI ČÍTAČI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATEJ TURCEL**
AUTOR PRÁCE

SUPERVISOR **Mgr. LUKÁŠ HOLÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student:	**Turcel Matej, Bc.**
Programme:	Information Technology
Field of study:	Mathematical Methods in Information Technology
Title:	**Minimization of Counting Automata**
Category:	Algorithms and Data Structures

Assignment:

1. Familiarise yourself with the counting automata and counting-set automata used in [1] for pattern matching of regular expressions with bounded repetition, and with algorithms for simulation based reduction of non-deterministic automata.
2. Propose means of minimization/size reduction for counting and/or counting-set automata, based on generalizations of the principle of deterministic minimization and simulation based reduction.
3. Implement your size reduction algorithm.
4. Evaluate the reduction capabilities of your algorithm and its efficiency against automata from regular expressions appearing in practice.

Recommended literature:

1. Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex Matching with Counting-Set Automata . Proc. ACM Program. Lang. 4, OOPSLA, Article 218 (November 2020), 30 pages. https://doi.org/10.1145/3428286

Requirements for the semestral defence:

1. Item 1 of the assignment,
2. initialization of the work on 2,
3. at least a small part of the text of the thesis.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Holík Lukáš, Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	March 9, 2021

Abstract

This works deals with size reduction of *counting automata (CA)*. Counting automata extend the classical finite automata with bounded counters. This allows efficient handling of e.g. regular expressions with repetition: `a{5,10}`. In this thesis we discusses the simulation relation in CA, which allows us to reduce their size. We rely on classical simulation in finite automata, which we non-trivially extend to CA. The key difference lies in the necessity to simulate counters as well as states. To this end, we present the novel concept of *parameterized simulation relation* in CA, and propose methods for computing this relation and using it to reduce the size of a CA. The proposed methods have been implemented and their efficiency experimentally evaluated.

Abstrakt

Táto práca sa zaoberá redukciou veľkosti tzv. *čítačových automatov*. Čítačové automaty rozširujú klasické konečné automaty o čítače s obmedzeným rozsahom hodnôt. Umožňujú tým efektívne spracovať napr. regulárne výrazy s opakovaním: `a{5,10}`. V tejto práci sa zaoberáme reláciou simulácie v čítačových automatoch, pomocou ktorej sme schopní zredukovať ich veľkosť. Opierame sa pritom o klasickú simuláciu v konečných automatoch, ktorú netriviálnym spôsobom rozširujeme na čítačové automaty. Kľúčovým rozdielom je nutnosť simulovať okrem stavov taktiež čítače. Za týmto účelom zavádzame nový koncept *parametrizovanej relácie simulácie*, a navrhujeme metódy výpočtu tejto relácie a redukcie veľkosti čítačových automatov pomocou nej. Navrhnuté metódy sú tiež implementované a je vyhodnotená ich efektivita.

Keywords

counting automaton, simulation, reduction of automata

Klíčová slova

čítačový automat, simulácia, redukcia automatov

Reference

TURCEL, Matej. *Minimization of Counting Automata*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Lukáš Holík, Ph.D.

Rozšířený abstrakt

V tejto práci študujeme simuláciu v klasických čítačových automatoch za účelom redukcie ich veľkosti. Cieľom práce je rozšíriť tieto metódy na tzv. *čítačové automaty*, zavedené v práci [55]. Toto rozšírenie je do značnej miery netriviálne. Pokiaľ je nám známe, doposiaľ neboli navrhnuté podobné rozšírenia minimalizačných metód na čítačové automaty alebo iné modely automatov im podobné.

Prvým hlavným prínosom tejto práce je nový koncept *parametrizovanej relácie simulácie* na čítačových automatoch. Tento koncept formálne definujeme a zvolenú definíciu odôvodňujeme. V analýze jej vlastností ukazujeme, že na rozdiel od klasických konečných automatov, čítačové automaty môžu mať viacero simulačných *preorderov*. Ďalej tiež ukazujeme, že čítačové automaty nemôžu byť redukované pomocou ľubovoľnej simulácie; táto simulácia musí splňovať určité podmienky. Toto nás motivovalo k zavedeniu pojmu *konzistentnej simulácie* – simulácie, ktorá môže byť použitá na redukciu automatu.

Ďalej navrhujeme metódu redukcie čítačových automatov pomocou konzistentnej simulácie, a prezentujeme náznak dôkazu korektnosti tejto redukcie. Keďže pôvodný formalizmus čítačových automatov nám neumožňuje aplikovať túto redukciu, rozširujeme formalizmus o operácie premenovania čítačov na prechodoch.

Druhým hlavným prínosom tejto práce je algoritmus na výpočet parametrizovanej relácie simulácie na čítačových automatoch. Tento algoritmus pozostáva z dvoch častí, ktoré spoločne nájdu všetky konzistentné simulácie v automate. Algoritmus sme implementovali a experimentálne sme vyhodnotili jeho redukčné schopnosti. Použili sme pritom rozsiahlu databázu regulárnych výrazov, s ktorej sme využili na vyhodnotenie vyše 28000 výrazov s opakovaním. Vzhľadom k tomu, že použitá metóda prekladu regulárnych výrazov na čítačové automaty produkuje automaty blízke optimálnym, boli dosiahnuté redukcie pomerne skromné. Za daných okolností sú ale výsledky uspokojivé, a súdime, že nami prezentovaná metóda redukcie má potenciál praktického využitia.

Minimization of Counting Automata

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mgr. Lukáš Holík, PhD. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Matej Turcel
May 19, 2021

Acknowledgements

I would like to thank my supervisor Mgr. Lukáš Holík, PhD. for his guidance and support during my work on this thesis. I also appreciate the help of Ing. Lenka Turoňová, who introduced me to the CountingAutomata library.

Contents

1 Introduction **3**

2 Preliminaries **7**
2.1 Miscellaneous Conventions 7
2.2 Finite Automata (FA) 7
2.3 NFA Reduction Algorithms 9
2.3.1 Simulation Preorder in FA 10
2.3.2 Reducing NFA Using the Simulation Preorder 11
2.3.3 Ilie–Navarro–Yu (INY) Algorithm 13
2.4 Counting Automata 15
2.4.1 Effective Boolean Algebras 15
2.4.2 Words and Regexes over Effective Boolean Algebras 15
2.4.3 Minterms 16
2.4.4 Symbolic Automata 16
2.4.5 Counting Automata (CA) 17

3 Simulation in CA **22**
3.1 Counter Liveness 22
3.2 Counter Mapping 23
3.3 Definition of CA Simulation 27
3.4 Correctness of Parameterized Simulation 32
3.5 Hypersimulation and its Properties 33

4 Simulation-Based Reduction of CA **41**
4.1 Correctness of Merging 45

5 Proposed Algorithm for Computing Simulations in CA **47**
5.1 Algorithm Pseudosim: Computing the Pseudosimulation 48
5.1.1 Initialization Phase 50
5.1.2 Inductive Phase 55
5.2 Algorithm Search: Searching for Consistent Simulations in the Pseudosimulation 59
5.2.1 Reduction to SAT 62

6 Experimental Evaluation **65**
6.1 Environment 65
6.2 Input Data 66
6.3 Experiments 66

6.3.1 Experiment 1: Individual Regexes 67
6.3.2 Experiments 2 and 3: Disjunctions of Regexes 68
6.4 Discussion of Results 71

7 Conclusions and Future Work **72**

Bibliography **74**

Appendices **80**

A Left Simulation in CA **81**

B Upper Bounds of Complexity of Algorithm Pseudosim and Algorithm Search **83**
B.1 Time and Space Complexity of Algorithm Pseudosim 83
B.2 Size of SAT Formulation in Algorithm Search 84

C Computing Bisimulations **85**
C.1 Algorithm Pseudosim 85
C.1.1 Initialization Phase 85
C.1.2 Inductive Phase 86
C.2 Algorithm Search 87

D Additional Figures from Experiments **88**

Chapter 1

Introduction

Our work deals with minimization of nondeterministic *counting automata*. The primary area of application for counting automata is regular expression matching, with prospective future applications in verification and decision procedures of logics. The task of matching regular expressions (regexes) arises very frequently, in many different areas, and is needed for many different purposes. Filtering, finding and replacing patterns in text, and data validation are among the most prominent applications. Based on prior studies, approximately 30-40% of software written in Java, JavaScript and Python uses regex matching [20].

Regex matching. Textbook algorithms for regex matching rely on using deterministic finite automata (DFA). When matching using a DFA, a single input character is processed in constant time with respect to the size of the automaton. Thus, the input string is processed in time at most linear in the length of the string. However, the finite automata (FA) obtained from a regex (RE) via classical RE-to-FA construction [41] are generally nondeterministic (NFA). A determinization of the NFA is necessary in order to use it for matching, and this determinization can take exponential time in the size of the input regex.

An exponential blow-up often occurs as a result of use of the *counting operator*, which expresses a repeated regex match with a lower and upper bound on the number of repetitions. For example, the regex `[ab]{2,5}` matches the string `abab`. An example of a regex which causes exponential blowup is `.*a.{`k`}`. It matches any string, such that its kth-to-last symbol is `a`. Using a classical RE-to-NFA construction, we obtain a NFA with $\Theta(k)$ states, and its determinization yields a DFA with $\Theta(2^k)$ states. This exponential blowup is necessary: to ensure that the kth-to-last symbol is `a`, we need to remember for each of the last k seen symbols whether it is `a`. This requires a single bit per symbol; k bits in total. The number of possible values encoded on k bits is 2^k. Therefore, at least 2^k states are needed to distinguish all possibilities.

Due to the exponential cost of a priori determinization, regex matchers tend to avoid it. Avoiding determinization, however, comes at the cost of increased matching complexity. A particularly common algorithm of this kind is Spencer's backtracking algorithm [52], which operates directly on a NFA. It works by backtracking on the input string, so the input is possibly read multiple times. Due to this fact, matching can take time exponential in the *length of the input string*. This may be much worse than determinization, which is exponential in the size of the regex.

More sophisticated algorithms avoid exponential a priori cost by utilizing on-the-fly determinization [53], while maintaining matching time linear in the input length. They are employed by many state-of-the-art regex matchers, such as GNU grep [27], in combination

with caching to avoid re-computing already encountered sets of states. However, on regexes such as `.*a.{`k`}`, these algorithms match a single character in time at worst linear in k, and due to the size of the state space, may be ineffective even with caching.

Counting automata. To mitigate the possibility of an exponential blow-up due to use of the counting operator, numerous works (e.g. [25; 51]) have proposed extensions of classical NFA which address this problem. In [25], classical NFA are extended by *bounded counters*. Recently, the work [55] generalized this approach to *symbolic finite automata* (SFA). Such extended SFA are called *counting automata* (CA), and their deterministic counterparts are called *counting-set automata* (CsA). Counters encode the number of repetitions seen so far; for example, a CA corresponding to the regex `a{2,5}` uses a counter with lower bound 2 and upper bound 5. CsA maintains counting sets instead of counters, which encode several possible values of a counter. The basic principle of CsA is that of the on-the-fly determinization of [53] – postponing the most costly part of determinization until runtime.

Due to the use of counters, the aforementioned regex `.*a.{`k`}` no longer yields a nondeterministic automaton with $\Theta(k)$ states, but only $\Theta(1)$ states. Furthermore, the number of states remains $\Theta(1)$ after determinization. In general, the determinization does not suffer from exponential explosion as the textbook algorithm [46] does.

In the deterministic CsA obtained from `.*a.{`k`}`, the values of past k symbols are encoded by a counting set: for each value j in the set, the jth-to-last symbol of the so-far consumed input is `a`. Naturally, we still need at least k bits to represent whether each of the past k symbols is `a`, but now the matching can be performed much more efficiently. Namely, upon reading an input character, every counting set can be updated in constant time. Therefore, the time complexity of processing a single input character is at most linear in the number of counters and does not depend on the counter bounds.

The possible uses of counting automata, however, are not limited to regex matching. Among the most promising are applications in formal verification; in particular model checking, where automata are utilized frequently. For example, string solving (model checking on programs manipulating strings) often requires working with numerical constraints, for example on string length, which could be represented using counting automata (e.g. [2; 1; 14]). Recently, the work [50] has presented the construction of (a variation of) CA from formulae of linear temporal logic with bounded repetition [50]. Another application is in verification of programs with bounded loops [28], which could be modeled by CA. Lastly, CA could be utilized in decision procedures of WS1S and WSkS logics [24] and Presburger arithmetic [58; 5].

Simulation-based reduction. When working with finite automata, it is often desirable to reduce their size in order to make operations on them more efficient. Traditional minimization algorithms such as [32] take a DFA as input and produce minimal equivalent DFA as output. Their general principle is merging states which are indistinguishable, meaning that the same set of words is accepted from both states. The requirement for complete a priori determinization makes such algorithms impractical, due to the aforementioned possibility of an exponential blowup. This motivates the development of *NFA reduction algorithms* [37; 9], which operate directly on nondeterministic automata, and can significantly reduce their size while having polynomial runtime. NFA reduction algorithms can be seen as generalizations of classical minimization algorithms, since they also generally operate by merging "indistinguishable" states. However, as described in further chapters, there are several notions of indistinguishability in NFA, which open doors for multiple possibilities of reduction.

One example of a simulation-reduction algorithm is the Ilie–Navarro–Yu algorithm [36], which builds on the earlier work in this area [29; 37; 12]. It works by computing the *simulation preorder* on the states of a NFA, which underapproximates *language inclusion*. State p is greater or equal to state q if its language is a superset of the language of q; in other words, if it accepts at least all the words q accepts. When both p is greater or equal to q and vice versa, then these states have equal language, and thus are indistinguishable and can be merged.

One may ask why should we not use the precise language inclusion, why do we need its approximation. Indeed, ideally we would like to merge *all* language-equivalent states, not only those which are equivalent according to the simulation preorder. However, the major drawback of the exact approach is its computational complexity. Computing precise language inclusion in NFA is PSPACE-complete, while the simulation preorder can be computed in polynomial time.

Simulation in counting automata. This work extends the classical NFA reduction algorithms to CA. The CA formalism is in itself considerably more complex than that of FA. Its intricacy stems from the fact that the state space is not entirely known beforehand, but is "explored" dynamically during runtime, as the counter values change. The states of CA are thus "parameterized" by counter values. Therefore, when statically reasoning about CA, one must take into consideration the possible runtime configurations, which are determined by a combination of state and counter values. Because of these intricacies, it is desirable to make as many simplifications as possible in order to make the reasoning manageable. That is why the Ilie–Navarro–Yu algorithm was chosen as a candidate, being among the simplest known NFA reduction algorithms. Among more sophisticated algorithms for computing the simulation preorder are Ranzato–Tapparo [47], which achieves better performance by operating on equivalence classes instead of individual states, and the asymptotically best known algorithms presented in [17] and [15].

Owing to the parametric nature of CA states, the concept of simulation preorder in CA is not as straightforward as in NFA. Since the language of a state depends on specific counter values, we must find a certain correspondence between counters to show that two states are indistinguishable. The states must have counters which are compatible, in the sense that the same operations can be performed on them in their respective states. The simulation preorder is therefore parameterized by this correspondence, which we call *counter mapping*. In the most general case, there can be several mappings between a single pair of states. As a consequence, a single counting automaton can have multiple simulation preorders (in contrast, a NFA has exactly one simulation preorder). Such cases, however, appear to be very rare in CA arising from regular expressions.

Contributions and outline of this work. In this thesis, we make the following contributions. Firstly, we introduce the novel notion of parameterized simulation relation on counting automata (Chapter 3). Secondly, we propose a means of reducing the size of a CA based on this relation (Chapter 4). To this end, we extend the CA formalism by introducing a new *counter rename* operation, which is necessary for our merging reduction. Thirdly, we propose an algorithm to compute the simulation relation (Chapter 5). Lastly, we experimentally evaluate the reduction capabilities of our algorithm on CA obtained from various real-world regular expressions with counting (Chapter 6).

Related work. Simulation on classical FA and similar formalisms (such as Kripke structures [40]) has been studied extensively; e.g. in [29; 36; 47; 15]. However, to our knowledge, adoption of these methods to extended models of FA, such as counting automata, has not been attempted thus far. Neither the work of Turoňová et al. [55] which introduces counting automata, nor the work of Gelade et al. [25] on which the former is based, discusses minimalism of the considered automata, methods of their size reduction, or an analogue of simulation relation on them. The works of Hovland [34; 35] present a model of automata similar to CA, but again do not discuss minimalism, size reduction, or simulation.

Chapter 2

Preliminaries

This chapter introduces the theory used in the following chapters. Firstly, we define classical finite automata and their properties such as determinism, minimality and the left and right language of a state. We then define the simulation preorder on states of finite automata, and explain how it can be used to reduce the size of automaton. Lastly, we introduce the concept of counting automata (CA).

2.1 Miscellaneous Conventions

Domain and image of a function. Given a partial function $f\colon X \rightharpoonup Y$, we denote by $\mathbf{dom}\, f$ the domain of f ($\mathbf{dom}\, f \subseteq X$) and by $\mathbf{im}\, f$ the image of f ($\mathbf{im}\, f \subseteq Y$).

Constant predicates. In the context of Boolean predicates, we define the following constant predicates for any domain: $\mathbf{true} \stackrel{\text{def}}{=} \boldsymbol{\lambda} x\,.\,\top$ (the always-true predicate); $\mathbf{false} \stackrel{\text{def}}{=} \boldsymbol{\lambda} x\,.\,\bot$ (the always-false predicate). Here, $\top$ and $\bot$ stand for the truth values *true* and *false*, or 0 and 1, the precise meaning of which will be clear in the context.

2.2 Finite Automata (FA)

In this section we introduce the classical theory of finite automata [33]. We define the concept of intermediate language, which is not presented in literature, but which helps us in definitions of left and right language [12] and the language of a finite automaton. Following [18] and [19] and adapting as appropriate, we define properties of automata and their states.

An *alphabet* Σ is a finite non-empty set of *symbols*. A finite, possibly empty sequence of symbols $w = a_1 a_2 \dots a_n$, where $a_i \in \Sigma$, is referred to as *word* or *string* of length n over the alphabet Σ; the length of a word w is denoted $|w|$. The symbol $\boldsymbol{\varepsilon}$ denotes the empty word (word of length zero). The set of all words over the alphabet Σ is denoted Σ^*. A set of words over Σ, $\mathcal{L} \subseteq \Sigma^*$, is called a *language*. For two words $a = a_1 \dots a_n$ and $b = b_1 \dots b_m$ we denote by $a \cdot b$ the *concatenation* of a and b: $a_1 \dots a_n b_1 \dots b_m$. We define the concatenation of languages as $\mathcal{L}_1 \cdot \mathcal{L}_2 = \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}$.

Definition 2.2.1 (**Finite automaton**)**.** A *finite automaton (FA for short)* is a quintuple $M = (Q, \Sigma, \Delta, I, F)$, where

- Q is a finite set of *states*,

- Σ is a finite *alphabet*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*,
- $I \subseteq Q$ is the set of *initial states*,
- $F \subseteq Q$ is the set of *final states*.

A *transition* $(q, a, r) \in \Delta$ is also denoted $q \text{-}\langle a \rangle\!\!\rightarrow r$. We say that r is an *a-successor* of q, and q is *a-predecessor* of r. The symbol τ will be commonly used to refer to a transition; that is, whenever τ is used, it is assumed that $\tau \in \Delta$. We will sometimes use $q \text{-}\langle a \rangle\!\!\rightarrow r$ as a formula of predicate logic, with the meaning $\exists\, q \text{-}\langle a \rangle\!\!\rightarrow r \in \Delta$. Unless explicitly specified otherwise, we denote by n the number of states and by m the number of transitions of an automaton; and by ℓ the size of the alphabet.

We define the *reverse transition relation* $\overleftarrow{\Delta} = \{r \text{-}\langle a \rangle\!\!\rightarrow q \mid q \text{-}\langle a \rangle\!\!\rightarrow r \in \Delta\}$. To avoid ambiguity, we sometimes use the symbol $\overrightarrow{\Delta}$ as a synonym of Δ.

The *reverse automaton* for automaton $M = (Q, \Sigma, \overrightarrow{\Delta}, I, F)$ is $M^R = (Q, \Sigma, \overleftarrow{\Delta}, F, I)$.

For $a \in \Sigma$, we define the *a-restricted transition relation* $\overrightarrow{\Delta}_a \subseteq Q \times Q$ where $r \in \overrightarrow{\Delta}_a(q)$ iff $q \text{-}\langle a \rangle\!\!\rightarrow r \in \overrightarrow{\Delta}$. In an analogous fashion, we define the *reverse a-restricted transition relation* $\overleftarrow{\Delta}_a \subseteq Q \times Q$, where $r \in \overleftarrow{\Delta}_a(q)$ iff $r \text{-}\langle a \rangle\!\!\rightarrow q \in \overleftarrow{\Delta}$. We lift the restricted transition relations to sets of states: $\Delta_a(S) = \bigcup_{s \in S} \Delta_a(s)$.

The *extended restricted transition relation*, denoted the same way as the ordinary restricted transition relation, is defined as $\Delta_{aw} = \Delta_w \circ \Delta_a$, where Δ_ε is the identity relation.[1] Extended restricted transition relations are also lifted to sets of states.

Let $w = a_1 \dots a_n$ be a word of length n and $q, r \in Q$ be states of a FA M. We say that w *transfers* M from q to r iff there exists a sequence of transitions $(\tau_i = p_i \text{-}\langle a_i \rangle\!\!\rightarrow p'_i)$ for $i \in [1, n]$, such that $p_1 = q$, $p'_n = r$, and $p'_i = p_{i+1}$ for each $i \in [1, n-1]$.

A word w is *accepted* from state q in M iff w transfers M from q to some $f \in F$, and w *leads* to q iff w transfers M from some $i \in I$ to q.

Definition 2.2.2 (**Intermediate language**)**.** The *intermediate language of states p and q* (or simply *language between p and q*) is defined as $\mathcal{L}_B(p, q) = \{w \in \Sigma^* \mid q \in \Delta_w(p)\}$. We further lift the definition to sets of states: $\mathcal{L}_B(S, T) = \bigcup_{s \in S,\, t \in T} \mathcal{L}_B(s, t)$, and similarly if only one of the arguments is a set. The language between states p and q contains precisely those words which transfer M from p to q.

Definition 2.2.3 (**Left language of a state**)**.** The *left language of a state q* is defined as $\mathcal{L}_L(q) = \mathcal{L}_B(I, q)$. We lift the definition to sets of states: $\mathcal{L}_L(S) = \mathcal{L}_B(I, S)$. The left language of a state q contains precisely those words which lead to q.

Definition 2.2.4 (**Right language of a state**)**.** The *right language of a state q* (or simply *language of q*) is defined as $\mathcal{L}_R(q) = \mathcal{L}(q) = \mathcal{L}_B(q, F)$. We lift the definition to sets of states: $\mathcal{L}_R(S) = \mathcal{L}_B(S, F)$. The right language of a state q contains precisely those words which are accepted from q.

Definition 2.2.5 (**Language of FA**)**.** The *language of a FA M*, $\mathcal{L}(M)$, is the language between the initial and final states of M: $\mathcal{L}(M) = \mathcal{L}_B(I, F)$. Two FA are *equivalent* iff their languages are equal. For some word $w \in \Sigma^*$, M *accepts* or *recognizes* w iff $w \in \mathcal{L}(M)$.

[1] In automata with ε-transitions (where the alphabet symbol on a transition may be the empty word ε), it would not be the identity relation but the ε-closure of the given state (the set of all states reachable from this state via zero or more ε-transitions).

Properties of states of FA. A state q of FA is said to be:

- *unreachable* iff $\mathcal{L}_L(q) = \emptyset$, *reachable* otherwise,
- *rejecting* or *dead* iff $\mathcal{L}_R(q) = \emptyset$, *nonrejecting* otherwise,
- *complete* iff $\forall a \in \Sigma\colon \Delta_a(q) \neq \emptyset$, *partial* otherwise,
- *deterministic* iff $\forall a \in \Sigma\colon |\Delta_a(q)| \leq 1$, *nondeterministic* otherwise.

Properties of FA. A FA M is said to be:

- *clean* iff all states of M are reachable,
- *trim* iff it is clean and has no rejecting states,
- *complete* iff all states of M are complete, *partial* otherwise,
- *deterministic* iff $|I| \leq 1$ and all states are deterministic; *nondeterministic* otherwise.

We use the abbreviations *DFA* and *NFA* to refer to a deterministic and nondeterministic FA respectively.

We note that every FA can be made complete in time $O(\ell n)$ and every NFA can be made deterministic in time $O(2^n)$ [46; 43].

Definition 2.2.6 (**Minimality of DFA**)**.** A partial (or complete) DFA M is *minimal* iff no equivalent partial (or complete) DFA with fewer states exists. Equivalently, M is clean and no two distinct states have equal right language; and if M is partial, then it is trim. (Adapted after [18].)

Definition 2.2.7 (**Minimality of NFA**)**.** A NFA M is *Q-minimal* or simply *minimal* iff there exists no equivalent NFA with fewer states. M is *Δ-minimal* iff there exists no equivalent NFA with fewer transitions. (Adapted after [19].)

We note that while each FA has a *unique* equivalent minimal DFA (whether complete or partial), the minimal equivalent NFA is in general not unique (for an example, see [4]). We further note that while the minimal DFA for a given input DFA can be found in time $O(\ell n \log n)$ [32], finding a minimal NFA for a given FA is PSPACE-complete and thus unlikely to be solvable in polynomial time [39]. This problem cannot even be efficiently approximated [26].

2.3 NFA Reduction Algorithms

In order to devise a reduction algorithm for counting automata, we shall first examine existing methods of reduction of nondeterministic finite automata. In this section, we describe the necessary concepts and basic methods of NFA reduction. These methods will serve as a guideline in the design of our new reduction algorithm for counting automata.

In practice, one frequently deals with nondeterministic FA obtained by some conversion algorithm, such as converting regexes to NFA. These conversion algorithms often produce NFA which exhibit certain redundancy, in the sense that several states behave the same. For example, assume a FA with two reachable states p, q, which are both final and have no

outgoing transitions. Clearly, they have equal right language – $\{\varepsilon\}$. We lose no information by replacing these two states by a single final state r, and redirecting the transitions into p and q to go into r instead.[2] By doing this, the size of the FA (in this case, the number of states) is reduced, while preserving its original language.

Classical algorithms for reducing the size of FA, such as the well-known algorithm of Hopcroft [32], operate on deterministic FA. Due to the possibly exponential cost of determinization, it is desirable to avoid it and reduce a NFA directly. That is why *polynomial-time NFA reduction algorithms* have been studied, as means to reduce the size of an automaton without resorting to determinization [37; 38; 12]. In many cases, the reduced NFA would still be determinized eventually. However, its reduced size can make the determinization significantly more efficient, as the savings in NFA size can project into exponentially larger savings in the cost of determinization. We will describe one such reduction algorithm, based on the concept of *simulation preorder*.

Simulation preorder is an underapproximation of language inclusion on states of NFA. Unlike true language inclusion, which is PSPACE-complete for NFA [21], the simulation preorder can be computed in polynomial time [36]. Owing to this fact, NFA simulation is also frequently exploited in other areas than NFA reduction. In formal verification, for example, deciding language inclusion, equivalence, or universality plays a key role [3]. Simulation can be used to accelerate all of these decision problems.

2.3.1 Simulation Preorder in FA

This section gives definitions of simulation relation, simulation preorder and related concepts, mostly in accordance with [38], [12] and [36].

Definition 2.3.1 (**Simulation relation**)**.** A *right simulation* (or simply *simulation*) on FA M is a binary relation $S_R \subseteq Q \times Q$, such that for each (q, s) in this relation, the following conditions hold:

$$q \in F \Rightarrow s \in F, \tag{2.1}$$

$$\forall\, q \trans{a} q' \in \Delta \;\; \exists\, s \trans{a} s' \in \Delta\colon \;\; (q', s') \in S_R. \tag{2.2}$$

We usually drop the *right* specifier and use simply *simulation*, *simulates*, etc. to refer to the right simulation. Observe that while Condition 2.1 is necessary for $\mathcal{L}_R(q) \subseteq \mathcal{L}_R(s)$ to hold, Condition 2.2 is overly strict. For example, if q has a single a-successor q' which is a dead state, then s does not need to have an a-successor in order for $\mathcal{L}_R(q) \subseteq \mathcal{L}_R(s)$ to hold.

It is easy to see that any state can simulate itself, and thus the reflexive closure of a simulation is a simulation. The transitive closure of a simulation is also a simulation, and so is the union of two simulations. From this it follows that there exists a unique maximal simulation, which is reflexive and transitive; hence a *preorder* [42].

Definition 2.3.2 (**Simulation preorder**)**.** Let S be a simulation relation on a FA M. If there exists no simulation S' on M, s.t. $S \subset S'$, then S is the *(right) simulation preorder* of M, denoted $\preccurlyeq$ (or $\preccurlyeq_R$).[3]

[2] This is true only under the assumption that the specific final state, in which the automaton accepts the input, bears no significance. We note that this is a standard assumption in related literature, albeit usually not mentioned explicitly.

[3] It should be noted that $\preccurlyeq$ is not necessarily the only simulation preorder on M. We use the term *simulation preorder* to refer to the *maximal* simulation preorder.

For $(q, s) \in \preccurlyeq_R$, we write $q \preccurlyeq_R s$ and say s *(right)-simulates* q; s is *stronger* than q and q is *weaker* than s. Simulation implies language inclusion – if state s simulates state q, then s accepts at least all the words q accepts (see [12] for proof):

$$q \preccurlyeq s \implies \mathcal{L}_R(q) \subseteq \mathcal{L}_R(s). \tag{2.3}$$

Bisimulation and bisimulation equivalence. A simulation that is symmetric is called *bisimulation*. It follows that the unique maximal bisimulation is an equivalence, called the *bisimulation equivalence*.

Simulation equivalence. Taking the maximal symmetric subset of $\preccurlyeq$ gives us an equivalence, called the *simulation equivalence*, which is not necessarily a simulation. We refer to this equivalence as $\preccurlyeq$*-equivalence*; two states belonging to it are said to be $\preccurlyeq$*-equivalent*. Observe that if $p \preccurlyeq q$ and $q \preccurlyeq p$, then $\mathcal{L}(p) = \mathcal{L}(q)$. Therefore, $\preccurlyeq$-equivalence partitions states into classes in which language equivalence holds. Put another way, it is a refinement of the language equivalence relation, and it is (in some cases[4]) strictly coarser than bisimulation equivalence. An example of a FA where two states are simulation-equivalent but not bisimulating can be found in [12].

Left simulation S_L and left simulation preorder $\preccurlyeq_L$ are defined analogously. For each $(q, s) \in S_L$, the following holds:

$$q \in I \Rightarrow s \in I, \tag{2.4}$$

$$\forall\, q' \text{-}(a)\!\!\rightarrow q \in \Delta \;\; \exists\, s' \text{-}(a)\!\!\rightarrow s \in \Delta\colon \;\; (q', s') \in S_L. \tag{2.5}$$

Left simulation preorder is then the largest left simulation. If s left-simulates q, then every word leading to q also leads to s: $q \preccurlyeq_L s \Rightarrow \mathcal{L}_L(q) \subseteq \mathcal{L}_L(s)$.

2.3.2 Reducing NFA Using the Simulation Preorder

In this section we describe how the size of a NFA can be reduced using the simulation preorder.

Merging states. By *merging a set of states* R of FA M, we understand modifying M by:

1. Creating a *new* state m in M (viz. $m \notin Q$ before the merging), such that
   (a) $\forall r \in R\colon \; q \text{-}(a)\!\!\rightarrow r \implies (\, q \text{-}(a)\!\!\rightarrow m \;\textbf{if}\; q \notin R \;\textbf{else}\; m \text{-}(a)\!\!\rightarrow m\,)$,
   (b) $\forall r \in R\colon \; r \text{-}(a)\!\!\rightarrow q \implies (\, m \text{-}(a)\!\!\rightarrow q \;\textbf{if}\; q \notin R \;\textbf{else}\; m \text{-}(a)\!\!\rightarrow m\,)$,
   (c) $R \cap I \neq \emptyset \implies m \in I$,
   (d) $R \cap F \neq \emptyset \implies m \in F$,
   (e) no other transitions into or from m exist.
2. Removing all states in R from Q, I, and F; and their coincident transitions from Δ.

As we have noted, two states with the same language can safely be merged, provided we only discern states by the language they accept. This observation has long been the basis of standard DFA minimization algorithms.

[4] Precisely in those cases, when it is not a simulation.

In a DFA, no word leads to two distinct states. As a consequence, the left simulation preorder is the identity relation and is of no use. In a NFA, however, the same word may lead to multiple states. As observed in [12], if two states have the same set of words which lead to them – the same left language – then we can merge them, much like we can merge states with equal right language. This already gives us two possibilities of reducing NFA using $\preccurlyeq_L$ and $\preccurlyeq_R$. We refer to merging a particular set of states based on their $\preccurlyeq_L$-equivalence as $\preccurlyeq_L$-*merging* or *left-merging*; reducing an automaton based on the $\preccurlyeq_L$ preorder is $\preccurlyeq_L$-*reduction* or *left-reduction*; and similarly for $\preccurlyeq_R$.

Maintaining preorder validity during merging. When using both $\preccurlyeq_L$ and $\preccurlyeq_R$ for merging states, it is necessary to realize that merging states according to one preorder might invalidate the other.

One possible solution is suggested in [12] (and erroneously transcribed in [36]). After $\preccurlyeq_R$-merging a set of states R into the new state m, we must remove from $\preccurlyeq_L$ the set $\{(r, s) \mid r \in R \wedge \exists r' \in R\colon r' \not\preccurlyeq_L s\}$. The merged states are also updated in $\preccurlyeq_L$ accordingly; i.e., all (r, s), where $r \in R, s \notin R$, are replaced by (m, s), and so on. In principle, as $\mathcal{L}_L(m)$ is the union of the left languages of all merged states, it can be left-simulated only by such s which simulates each of the merged states (and thus their "union", which is m).

Another solution to this problem, suggested by [13], is to first reduce an automaton according to one preorder, and only then compute and use the other preorder. These two steps may then be repeated as long as the automaton can be reduced.

Eliminating subsumed states. There is one more possibility of reduction given in [12]: merging two states where one "subsumes" the other. What we mean by "s subsumes q" is that both $\mathcal{L}_L(q) \subseteq \mathcal{L}_L(s)$ and $\mathcal{L}_R(q) \subseteq \mathcal{L}_R(s)$ holds. There is a caveat, however. Such merging might seem correct at a glance, but does not work if q is in a cycle labeled by word w and s is not in a cycle labeled by w.[5] The obvious solution – to simply delete the subsumed state – does not always work either. For examples for both of these cases, refer to [13]. We point out that in the example given therein for the latter case, q is subsumed by s, but $q \preccurlyeq_R s$ does not hold. In other words, the given example is contrived and does not arise in simulation-based reduction.

Little brother optimization. An optimization of simulation-based reduction is suggested by [9]. They propose, after merging equivalent states, to remove transitions to so-called *little brothers*. Assume $p \text{-}(a)\!\!\rightarrow q$, $p \text{-}(a)\!\!\rightarrow s$, $q \neq s$, and $q \preccurlyeq_R s$. Then q is a "little brother" of s, both being "children" of p. It follows immediately from $\mathcal{L}_R(q) \subseteq \mathcal{L}_R(s)$ that such transitions $p \text{-}(a)\!\!\rightarrow q$ can be removed without altering the language of p. After doing this, the automaton is cleaned (unreachable states are removed). The same can be applied for $\preccurlyeq_L$-reduction, removing transitions *from* little brothers per $\preccurlyeq_L$ (or rather, "little parents"), and then removing *dead* states.

[5] A state p being in a cycle c means c is a sequence of n transitions $(q_{i-1} \text{-}(a_i)\!\!\rightarrow q_i)$ for $0 < i \leq n$, s.t. $q_0 = q_n = p$. The label of c is the word $a_1 \dots a_n$.

2.3.3 Ilie–Navarro–Yu (INY) Algorithm

This section presents an algorithm for computing the simulation preorder, introduced by Ilie, Navarro and Yu [36], referred to as the *INY algorithm*. This algorithm is a modification of a similar, earlier algorithm [29], which operates on Kripke structures [40] instead of NFA. Compared to alternative algorithms such as that of Ranzato and Tapparo [47] or the more recent algorithms of [17] and [15], this algorithm is rather simple, yet maintains a favorable time complexity. We regard the simplicity of this algorithm as a crucial property for its successful adoption to CA, as it helps to make the already difficult reasoning about CA manageable.

As observed and proven in [22], the algorithm presented in the original publication [36] is applicable only to complete automata. We present a modified version of the algorithm which is applicable also to non-complete automata, as given in [30]. Algorithm 1 shows a pseudocode of the INY algorithm, computing the simulation preorder $\preccurlyeq$ of a FA M. For the sake of completeness, we now give a relatively detailed description of the INY algorithm. Readers may wish to proceed to the next section, as this description is mostly of supplementary nature and is not indispensable for understanding our new reduction algorithm for counting automata.

Algorithm 1: INY

Input: FA $M = (Q, \Sigma, \Delta, I, F)$
Output: The simulation preorder $\preccurlyeq$ of M

```
1  for p, q ∈ Q a ∈ Σ do
2  |   N_a(p, q) := |Δ⃗_a(p)|;
3  NotSimQueue := F × (Q \ F) ∪
4                 {(q, r) ∈ Q × Q | ∃ a ∈ Σ: Δ⃗_a(q) ≠ ∅ ∧ Δ⃗_a(r) = ∅};
5  Sim := Q × Q;
6  while NotSimQueue ≠ ∅ do
7  |   remove some (i, j) from NotSimQueue and Sim;
8  |   for a ∈ Σ do
9  |   |   for t ∈ Δ⃖_a(j) do
10 |   |   |   N_a(t, i) := N_a(t, i) − 1;
11 |   |   |   if N_a(t, i) = 0 then
12 |   |   |   |   for s ∈ Δ⃖_a(i) s.t. (s, t) ∈ Sim do
13 |   |   |   |   |   NotSimQueue := NotSimQueue ∪ (s, t);
14 return Sim;
```

An obvious approach is to compute $\preccurlyeq$ inductively, starting with an empty set and successively adding pairs of states while keeping the set a simulation. The problem with this naive approach can be seen easily: imagine some q in a cycle, simulated by some s. To show that s simulates q, its successors must simulate successors of q, and so on, and eventually we will come back to q and s. In other words, to show $q \preccurlyeq s$, we need to already know that $q \preccurlyeq s$. The simulation preorder therefore cannot be directly computed by induction. That is why the algorithm computes $\preccurlyeq$ *coinductively* [48] – it computes the complement of $\preccurlyeq$ inductively, and $\preccurlyeq$ is computed from this complement. Before we describe the details, let us first explain the algorithm on a higher level.

The algorithm maintains an overapproximation of $\preccurlyeq$, denoted *Sim*, which is initialized to $Q \times Q$. The complement of $\preccurlyeq$ is not computed and stored explicitly. Instead, non-simulating pairs (viz. (q, r) where $q \not\preccurlyeq r$) are removed from *Sim* as the algorithm proceeds. The queue *NotSimQueue* is used to store non-simulating pairs which have not been processed yet.[6] Each of these pairs is then removed from both *NotSimQueue* and *Sim* (line 7). Eventually, all pairs (q, r) s.t. $q \not\preccurlyeq r$ will go through *NotSimQueue*,[7] and will be removed from *Sim*. In the end, *Sim* will be $Q \times Q \setminus \{(q, r) \mid q \not\preccurlyeq r\} = \preccurlyeq$.

Initial approximation. The queue *NotSimQueue* is initialized to contain those pairs of states (q, r), of which we immediately know that $q \not\preccurlyeq r$. These are all those pairs, which *trivially* contradict the definition of simulation (Def. 2.3.1). Namely, no non-accepting state r can simulate an accepting state q, since $\varepsilon \in \mathcal{L}(q)$ but $\varepsilon \notin \mathcal{L}(r)$. Also, if q has some successor via a and r has no successor via a, then r trivially cannot simulate q. If the automaton is complete, however, it suffices to initialize *NotSimQueue* to $F \times (Q \setminus F)$ [22].

Counting of successors. The algorithm relies on the technique of *counting*. This technique is used to efficiently detect violation of Condition 2.2 in Definition 2.3.1. It works by keeping the number of successors of some state p via symbol a, which simulate q. This number is denoted $N_a(p, q)$ and is initialized to the total number of successors of p via a, for each p, a. As we discover new non-simulating pairs of states, we decrement the appropriate $N_a(p, q)$, such that it always corresponds to the number of simulating successors (according to the current approximation *Sim*). Whenever $N_a(p, q)$ reaches zero, we know from the definition of simulation that p cannot simulate any a-predecessor of q, say r. Therefore, we remove all such pairs (r, p) from *Sim*.

In the algorithm description, this is done on lines 10-13. Line 13 adds the pair to *NotSimQueue*, to be later removed from *Sim* on line 7. The check $(s, t) \in Sim$ on line 12 ensures that we do not add the same pair (s, t) to *NotSimQueue* twice. If it has already gone through *NotSimQueue*, then it is not in *Sim*.

Reusing the algorithm for computing $\preccurlyeq_L$. This algorithm, as presented, can be used without modification to compute $\preccurlyeq_L$. The key is to use the *reverse automaton* M^R of the original automaton M for which we wish to compute $\preccurlyeq_L$. This reverse automaton accepts the reversal of $\mathcal{L}(M)$, viz. $\mathcal{L}(M^R) = \{w^R \mid w \in \mathcal{L}(M)\}$ where $\varepsilon^R = \varepsilon$, $aw^R = w^R a$. Moreover, for each $q \in Q$, its left language in M equals its right language in M^R and vice versa. In fact, due to initial and final states being swapped between M and M^R, and transitions being reversed, $\preccurlyeq_R$ of M^R precisely coincides with $\preccurlyeq_L$ of M. This gives us a very straightforward way of computing $\preccurlyeq_L$ for some FA M: first compute M^R (trivial), then use the INY algorithm to compute $\preccurlyeq_R$ of M^R, which is $\preccurlyeq_L$ of M.

Complexity. The *time complexity* of the algorithm is $O(mn + \ell n^2)$. For the bound $O(\ell n^2)$, consider lines 1-2 which populate a 3-dimensional matrix of size $|\Sigma| \times |Q| \times |Q|$. This readily gives us also the *space complexity* – $\Theta(\ell n^2)$. Proof of the bound $O(mn)$ is more involved and can be found in appendix A of [31]. We note that in complete automata, $m \geq \ell n$, and thus the time bound simplifies to $O(mn)$.

[6] An important detail is that *NotSimQueue* an ordered set, meaning it cannot contain the same element twice. If this weren't the case, the loop on line 8 could be executed several times for a single non-simulating pair, yielding incorrect results.

[7] Having "gone through queue" means having been added to the queue and removed from it.

2.4 Counting Automata

In this section we describe the concept of *counting automata* recently introduced in [55]. Firstly, we define preliminary concepts such as effective Boolean algebras and symbolic finite automata, and using these, we give the definition of counting automata. The following definitions largely agree with those provided in [55]. We do not describe counting-set automata (CsA) here, as our work does not deal with them. For a detailed exposition, interested readers are referred to [55] or [56].

2.4.1 Effective Boolean Algebras

An *effective Boolean algebra* $\mathbb{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a *universe* of underlying domain elements. Ψ is a set of unary *predicates* closed under the Boolean connectives $\vee, \wedge \colon \Psi \times \Psi \to \Psi$ and $\neg : \Psi \to \Psi$; and $\bot, \top \in \Psi$ are the *false* and *true* predicates. Values of the algebra are sets of domain elements, and the *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ satisfies that $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\boldsymbol{Sat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$, and we say that φ is *satisfiable*. We require that $\boldsymbol{Sat}$ as well as $\vee$, $\wedge$, and $\neg$ are *computable* as a part of the definition of an effective Boolean algebra. We write $x \models \varphi$ for $x \in [\![\varphi]\!]$ and we use $\mathbb{A}$ as a subscript of a component when it is not clear from the context, e.g. $[\![\_]\!]_{\mathbb{A}} : \Psi_{\mathbb{A}} \to 2^{\mathfrak{D}_{\mathbb{A}}}$.

2.4.2 Words and Regexes over Effective Boolean Algebras

The basic building blocks of regexes are *predicates* from an effective Boolean algebra *CharClass* of *character classes*, such as the class of digits, written as `\d`. Let $\mathfrak{D} = \mathfrak{D}_{CharClass}$. A *word* over $\mathfrak{D}$ is a sequence of symbols $a_1 \cdots a_n \in \mathfrak{D}^*$ and a *language* $\mathcal{L}$ over $\mathfrak{D}$ is a subset of $\mathfrak{D}^*$. As with words over ordinary alphabets (defined in Section 2.2), we use $\boldsymbol{\varepsilon}$ to denote the empty word; the concatenation of words u and v is denoted $u \cdot v$ (abbreviated to uv) and is lifted to sets as usual. Furthermore, we write $\mathcal{L}^n$ for the n-th power of $\mathcal{L} \subseteq \mathfrak{D}^*$ with $\mathcal{L}^0 \stackrel{\text{def}}{=} \{\boldsymbol{\varepsilon}\}$ and $\mathcal{L}^{n+1} \stackrel{\text{def}}{=} \mathcal{L}^n \cdot \mathcal{L}$.

The abstract syntax of regexes is the following, with $\alpha \in \Psi_{CharClass}$ and n, m being integers such that $0 \leq n$, $0 < m$, and $n \leq m$:

$$\boldsymbol{\varepsilon} \quad \alpha \quad R_1 \cdot R_2 \quad R_1|R_2 \quad R\{n,m\} \quad R*$$

The semantics of a regex R is defined as a subset of $\mathfrak{D}^*$ in the following way: $\mathcal{L}(\alpha) \stackrel{\text{def}}{=} [\![\alpha]\!]$, $\mathcal{L}(\boldsymbol{\varepsilon}) \stackrel{\text{def}}{=} \{\boldsymbol{\varepsilon}\}$, $\mathcal{L}(R_1R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, $\mathcal{L}(R_1|R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R\{n,m\}) \stackrel{\text{def}}{=} \bigcup_{i=n}^{m} \mathcal{L}(R)^i$, and $\mathcal{L}(R*) \stackrel{\text{def}}{=} \mathcal{L}(R)^*$.

2.4.3 Minterms

Let $\mathit{Preds}(R)$ be the set of all predicates that occur in a regex R, and let $\mathit{Minterms}(R)$ denote the set of *minterms* of $\mathit{Preds}(R)$. Intuitively, $\mathit{Minterms}(R)$ is a set of non-overlapping predicates that can be treated as a concrete finite alphabet. Each minterm is essentially an indivisible region in the Venn diagram of the predicates in R: it is a satisfiable conjunction $\bigwedge_{\psi \in \mathit{Preds}(R)} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. For example, if $R =$ `[0-z]{4}[0-8]{5}`, then $\mathit{Preds}(R) = \{$`[0-8]`, `[0-z]`$\}$ and $\mathit{Minterms}(R) = \{$`[0-8]`, `[9-z]`, `[^0-z]`$\}$. Formally, if $\alpha \in \mathit{Minterms}(R)$, then $\boldsymbol{\mathit{Sat}}(\alpha)$ and $\forall \psi \in \mathit{Preds}(R)$: $[\![\alpha]\!] \cap [\![\psi]\!] \neq \emptyset \Rightarrow [\![\alpha]\!] \subseteq [\![\psi]\!]$.

Although the number of minterms of a general set X of predicates may be exponential in $|X|$, it is only linear if X consists of intervals, such as discrete intervals of symbols used in regexes – e.g. `[a-zA-Z]` denotes two intervals, and `[^a-zA-Z]` denotes their complement, which is equivalent to the union of three intervals. To witness, consider that the total number of unique border points (beginnings and ends) of n intervals is at most $2n$. The complement of a discrete (integer) interval $[a, b]$ requires 4 border points at most: $[-\infty, a-1]$, $[b+1, \infty]$. The total number of border points is thus linear in the number of input intervals. The set of minterms is the union of some of the intervals between two *neighboring* border points (meaning there is no other border point between them); namely those intervals which are covered by at least one input interval. There are $\Theta(n)$ pairs of neighboring points, and thus $O(n)$ minterms.

2.4.4 Symbolic Automata

Symbolic finite automata (SFAs) extend classical finite automata by having an alphabet given by an effective Boolean algebra. Formally, an SFA is a tuple $A = (\mathbb{I}, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the input algebra, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times Q$ is a finite set of transitions. A transition $(q, \alpha, r) \in \Delta$ will be also written as $q \text{-}(\alpha)\!\!\rightarrow r$.

A *run of A from a state p_0* over a word $a_1 \cdots a_n$ is a sequence of transitions $(p_{i-1} \text{-}(\alpha_i)\!\!\rightarrow p_i)$ for $1 \leq i \leq n$, with $a_i \in [\![\alpha_i]\!]$; the run is *accepting* if $p_n \in F$. The *language of A from a state q*, denoted $\mathcal{L}_A(q)$, is the set of words over which A has an accepting run from q. The *language of A*, denoted $\mathcal{L}(A)$, is $\mathcal{L}_A(q_0)$. A classical finite automaton can be understood as an SFA where the basic predicates have singleton set semantics, i.e., when for each concrete letter a there is a predicate α_a such that $[\![\alpha_a]\!] = \{a\}$.

A is *deterministic* iff for all $p \in Q$ and all transitions $p \text{-}(\alpha)\!\!\rightarrow q$ and $p \text{-}(\alpha')\!\!\rightarrow r$, if $\alpha \wedge \alpha'$ is satisfiable, then $q = r$. A is *mintermized* iff for all transitions $p \text{-}(\alpha)\!\!\rightarrow q$ and $p' \text{-}(\alpha')\!\!\rightarrow q'$, either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$. That is, different character predicates do not overlap and can be treated as plain symbols in a classical FA.

2.4.5 Counting Automata (CA)

Counting automata (CA) introduced in [55] extend classical NFA by bounded counters. They chiefly build upon the model of *counter NFA* introduced in [25] and generalize it to symbolic automata. CA are a natural automata model for regexes which use the counting operator. This is the main motivation for their use, since many regexes with counting induce an exponential blowup in the classical (S)FA model.

Counters count the number of passes through some part of the automaton, corresponding to a counted sub-expression of a regex. For example, given the regex `a{2,5}`, the corresponding CA uses a counter with lower bound 2 and upper bound 5. Counters are always bounded – every counter of a CA has a finite maximum value which it can attain during a run of the CA on any input. This requirement ensures that the state space of CA is finite, and CA thus have the same expressive power as ordinary FA. The counting operator, however, allows the maximum number of repetitions to be unbounded, e.g. `a{2,*}`. For such cases, the regex is translated into the equivalent form `a{2}a*`, which results in a CA with a single counter, having both lower and upper bound equal to 2.

We now formalize the notions of counters, their bounds, and operations performed on them. To this end, we introduce the concept of counter memory, which assigns values to counters during a CA run; and counter updates and update guards, which represent operations on counters and conditions under which these operations can be performed. Following these definitions, we formally define counting automata and their semantics. We note that some parts of our definitions differ from those in [55]. Namely, we completely refrain from using the concept of counter algebra, as the original definitions based on it do not provide the flexibility needed in further reasoning, presented in later chapters. We assert (without a proof) that our definitions are semantically equivalent with the original ones.

Counters, counter bounds. We denote by C a finite set of *counters*. A counter $c \in C$ is a unique object which has certain properties (e.g. its value) associated to it via functions. Two such functions are the *lower bound* function $\boldsymbol{min}\colon C \to \mathbb{N}$, and the *upper bound* function $\boldsymbol{max}\colon C \to \mathbb{N}$. For a counter c, we denote its lower and upper bounds $\boldsymbol{min}_c$ and $\boldsymbol{max}_c$ respectively. The following natural conditions holds for each counter $c \in C$: $\boldsymbol{min}_c \geq 0$, $\boldsymbol{max}_c > 0$, and $\boldsymbol{min}_c \leq \boldsymbol{max}_c$. We use the following notations, for any $n \in \mathbb{N},\ c, d \in C$:

$$
\begin{aligned}
n \sim c &\stackrel{\text{def}}{=} \boldsymbol{min}_c \leq n \leq \boldsymbol{max}_c, \\
c \subseteq d &\stackrel{\text{def}}{=} \forall\, n \in \mathbb{N}\colon\ n \sim c \Rightarrow n \sim d \\
&\equiv \boldsymbol{min}_c \geq \boldsymbol{min}_d \wedge \boldsymbol{max}_c \leq \boldsymbol{max}_d.
\end{aligned}
$$

When $c \subseteq d$, we say c is *subsumed* by d, or that *subsumption* of c by d holds.

Counter memory. A *counter memory* stores the current values of counters during a run of CA. It is a function $\mathfrak{m}\colon C \to \mathbb{N}$ which assigns to each counter c a value within its respective upper bound: $0 \leq \mathfrak{m}(c) \leq \boldsymbol{max}_c$ for all $c \in C$. Albeit its name may suggest that it records a history of counter values over time, this is not the case – only the current counter values are encoded by the memory. The set of all counter memories is denoted $\mathfrak{M}$. The *zero memory*, denoted $\mathfrak{m}_0$, is $\{c \mapsto 0 \mid c \in C\}$.

Counter updates and update guards. A *counter update* or simply *update* is one of the four functions $\{\textsc{noop}, \textsc{incr}, \textsc{exit}, \textsc{exit1}\}$, which have the form $\mathbb{N} \to \mathbb{N}$ and are defined as

$$\begin{array}{llll} \textsc{noop} & \stackrel{\text{def}}{=} & \boldsymbol{\lambda} n \,.\, n & \qquad \textsc{exit} \stackrel{\text{def}}{=} \boldsymbol{\lambda} n \,.\, 0 \\ \textsc{incr} & \stackrel{\text{def}}{=} & \boldsymbol{\lambda} n \,.\, n+1 & \qquad \textsc{exit1} \stackrel{\text{def}}{=} \boldsymbol{\lambda} n \,.\, 1 \end{array}$$

Counter updates occur on the transitions of a counting automaton, as explained later. They modify the values of counters during a CA run. To ensure that counter values do not exceed their respective upper bounds, we must employ update guards, which are associated with updates and which decide whether an update can be performed.

A *bound update guard*, also called *update guard of counter c* or simply *guard of c*, is one of the three predicates $\{\textsc{canIncr}_c, \textsc{canExit}_c, \textsc{canNoop}_c\}$, which have the form $\mathbb{N} \to \{\bot, \top\}$, and are bound to a specific counter c in the sense that they depend on the upper and lower bounds of c. Bound updates are defined as

$$\begin{array}{lll} \textsc{canIncr}_c & \stackrel{\text{def}}{=} & \boldsymbol{\lambda} n \,.\, n < \boldsymbol{max}_c \\ \textsc{canExit}_c & \stackrel{\text{def}}{=} & \boldsymbol{\lambda} n \,.\, n \geq \boldsymbol{min}_c \\ \textsc{canNoop}_c & \stackrel{\text{def}}{=} & \boldsymbol{\lambda} n \,.\, \top \end{array}$$

Like counter updates, bound update guards also occur on CA transitions. A bound guard decides, given the value of its counter c, whether an update can be performed on c. As the names suggest, each guard is associated with an update. In order to define this association formally, we first define unbound update guards.

An *unbound update guard* is one of the three functions $\{\textsc{canIncr}, \textsc{canExit}, \textsc{canNoop}\}$, which have the form $C \to (\mathbb{N} \to \{\bot, \top\})$, and which given a counter c produce the corresponding bound update guard for c, e.g. $\textsc{canIncr}(c) = \textsc{canIncr}_c$.

Each update is associated with an unbound guard (we say the update *induces* the guard) via the guard-of-update function $\mathcal{G}\colon [\mathbb{N} \to \mathbb{N}] \to [C \to (\mathbb{N} \to \{\bot, \top\})]$, defined as

$$\begin{array}{llll} \mathcal{G}(\textsc{noop}) & = \textsc{canNoop} & \qquad \mathcal{G}(\textsc{exit}) & = \textsc{canExit} \\ \mathcal{G}(\textsc{incr}) & = \textsc{canIncr} & \qquad \mathcal{G}(\textsc{exit1}) & = \textsc{canExit} \end{array}$$

Intuitively, the NOOP update does not modify the value of the counter and is always enabled. The operation INCR increments the counter and is enabled if the counter has not yet reached its upper bound. The operation EXIT resets the counter to 0 on exit from a counting loop and is enabled when the counter has reached its lower bound. The operation EXIT1 executes EXIT immediately followed by INCR.

We denote by $\textsc{Upd}$ the set of all counter updates, and by $\textsc{Grd}_C$ the set of all bound update guards for counters from the set C.

Definition 2.4.1 (Counting automaton). A *counting automaton (CA)* is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \mathit{fin}, \Delta)$, where

- $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*,
- C is a finite set of *counters*,
- Q is a finite set of *states*,
- $q_0 \in Q$ is the *initial state*,
- $F \subseteq Q$ is a finite set of *final states*,

- $\mathit{fin} \colon Q \to (C \to (\mathbb{N} \to \{\bot, \top\}))$ is the *acceptance condition*,
- $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times (C \to \textsc{Upd}) \times (C \to \textsc{Grd}_C) \times Q$ is the finite *transition relation*, where
  - an element of the set $\Psi_{\mathbb{I}}$ is the input predicate,
  - an element of $C \to \textsc{Upd}$ is a function assigning to each counter its update,
  - an element of $C \to \textsc{Grd}_C$ is a function assigning to each counter its bound guard.

We denote $\mathit{fin}(q)$ as fin_q. The acceptance condition of a particular state q and counter c is either $\textsc{canExit}_c$ or one of the constant predicates **true**, **false**. A state q is not in the set of final states F iff its acceptance condition is $\mathit{fin}_q = \{c \mapsto \textbf{false} \mid c \in C\}$. We postpone the explanation for having both a set of final states and an acceptance condition until we have defined the semantics of CA.

A transition $(p, \alpha, \mathit{upd}, \mathit{grd}, q) \in \Delta$ is denoted $p \text{-}(\alpha, \mathit{upd}, \mathit{grd})\!\!\rightarrow q$. To avoid ambiguity, we often use upd_τ and grd_τ to refer to the upd and grd components of the transition τ. For each transition $p \text{-}(\alpha, \mathit{upd}, \mathit{grd})\!\!\rightarrow q \in \Delta$, the following invariant holds for every counter $c \in C$: $\mathit{grd}(c) = \mathcal{G}(\mathit{upd}(c))(c)$. It is thus not necessary to include grd in the transition, since it can be obtained from upd. We include grd in the transition for convenience.

Example 2.4.1. Figure 2.1 shows a CA for the regex `.*a.{`k`}`. The counter c corresponds to the repetition `.{`k`}`, thus $\boldsymbol{\mathit{min}_c} = \boldsymbol{\mathit{max}_c} = k$. This CA is *mintermized*, which is defined further below and generally means the same as in SFA. On the transition from p to q, the first element a is the input predicate, the second element $\{c = 0\}$ is a *counter value assertion*. Such assertions can be identified by the surrounding curly braces. When present on a transition, the assertion expresses that whenever the transition is taken, the counter invariantly has the specified value. On the transitions in the state q, the second element $c{+}{+}$ is a shorthand for the counter operation $\textsc{incr}(c)$. The state q is final, although this is not depicted in the usual way (using a double circle). Instead, we write the acceptance condition ***fin*** near a state to indicate that it is final; otherwise it is not final. We also omit counters with the **true** condition. In the state s, the acceptance condition fin_s is $c \mapsto \textsc{canExit}_c$, symbolically written as *canExit(c)* in the diagram.

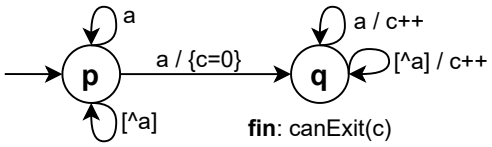


Figure 2.1: A counting automaton for the regex `.*a.{`k`}`.

Memory updates, guards, and acceptance conditions. To facilitate our definitions, we use the functions *memory update* $\mathfrak{upd}_\tau \colon \mathfrak{M} \to \mathfrak{M}$ and *memory guard* $\mathfrak{grd}_\tau \colon \mathfrak{M} \to \{\bot, \top\}$, which operate on counter memory. These two functions are associated with the transition τ, and are defined using the counter updates and update guards on this particular transition τ – upd_τ and grd_τ. We similarly define *memory acceptance condition* $\mathfrak{fin}_q \colon \mathfrak{M} \to \{\bot, \top\}$,

which is associated with the state q and defined using fin_q. They are defined as follows:

$$\mathfrak{upd}_\tau(\mathfrak{m}) = \{c \mapsto \mathit{upd}_\tau(c)(\mathfrak{m}(c)) \mid c \in C\},$$

$$\mathfrak{grd}_\tau(\mathfrak{m}) = \bigwedge_{c \in C} \mathit{grd}_\tau(c)(\mathfrak{m}(c)),$$

$$\mathfrak{fin}_q(\mathfrak{m}) = \bigwedge_{c \in C} \mathit{fin}_q(c)(\mathfrak{m}(c)).$$

Semantics of CA. The semantics of a CA A is defined through its *configuration automaton* $\mathit{SFA}(A)$; namely, its language $\mathcal{L}(A)$ is the language $\mathcal{L}(\mathit{SFA}(A))$. A *configuration* is a pair $\langle q, \mathfrak{m}\rangle \in Q \times \mathfrak{M}$ consisting of a state q and a counter memory $\mathfrak{m}$. The configuration automaton $\mathit{SFA}(A)$ of CA A an SFA whose states are A's configurations (there are finitely many of them), and the initial state of $\mathit{SFA}(A)$ is the *initial configuration* $\langle q_0, \mathfrak{m}_0\rangle$ of A. A state $\langle q, \mathfrak{m}\rangle$ of $\mathit{SFA}(A)$ is *final* iff $q \in F \wedge \mathfrak{fin}_q(\mathfrak{m})$ holds. The transition relation $\Delta_{\mathit{SFA}(A)}$ of $\mathit{SFA}(A)$ is defined as

$$\langle p, \mathfrak{m}\rangle \text{ -}(\alpha)\!\!\rightarrow \langle q, \mathfrak{m}'\rangle \in \Delta_{\mathit{SFA}(A)}$$
$$\textbf{iff}$$
$$\exists\, p \text{ -}(\alpha, \mathit{upd}_\tau, \mathit{grd}_\tau)\!\!\rightarrow q \in \Delta\colon\ \mathfrak{grd}_\tau(\mathfrak{m}) \ \wedge\ (\,\mathfrak{upd}_\tau(\mathfrak{m}) = \mathfrak{m}'\,).$$

We can now justify having both a set of final states and an acceptance condition. The reason is that we need to express three possibilities: a state of CA can be

1. *unconditionally non-final*, meaning it never accepts, regardless of counter values,
2. *conditionally final*, meaning it accepts conditionally, depending on counter values, or
3. *unconditionally final*, meaning it always accepts, regardless of counter values.

Using only a set of final states, we obviously could not express conditional finality; using only an acceptance condition, we could not express unconditional non-finality if the CA has no counters – the conjunction $\mathfrak{fin}_q(\mathfrak{m})$ is over an empty set of predicates and thus always holds.

Example 2.4.2. Figure 2.2 schematically depicts the configuration automaton for the CA in Fig. 2.1. In the diagram, the state labeled $\langle p, c = 0\rangle$ corresponds to the SFA state $\langle p, \mathfrak{m}\rangle$ where $\mathfrak{m}(c) = 0$; and likewise for other states. The state $\langle q, c = k\rangle$ is final; intermediate states between $\langle q, c = 1\rangle$ and $\langle q, c = k\rangle$ are omitted. As is clear from the diagram, this SFA has $\Theta(k)$ states, while the CA in Fig. 2.1 has $O(1)$ states with respect to k.

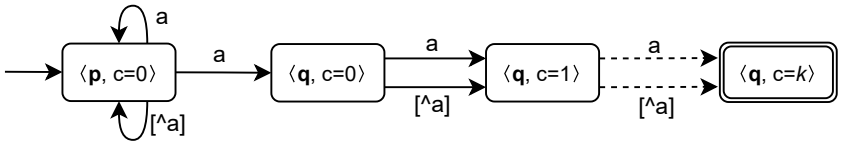


Figure 2.2: A configuration automaton of CA for the regex `.*a.{k}`.

Deterministic and simple CA. A counting automaton A is *deterministic* iff the following holds for its every state p and every two transitions $\tau_1 \colon p\text{-}(\alpha_1, grd_{\tau_1}, upd_{\tau_1})\!\!\rightarrow q_1$, $\tau_2 \colon p\text{-}(\alpha_2, grd_{\tau_2}, upd_{\tau_2})\!\!\rightarrow q_2$: if both $\alpha_1 \wedge \alpha_2$ and $\mathfrak{grd}_{\tau_1} \wedge \mathfrak{grd}_{\tau_2}$ are satisfiable, then $q_1 = q_2$ and $upd_{\tau_1} = upd_{\tau_2}$ (hence also $grd_{\tau_1} = grd_{\tau_2}$). It follows from the definitions that if A is deterministic, then $SFA(A)$ is deterministic too.

A is *simple* if all guards are satisfiable, and A is mintermized: for any two transitions $p\text{-}(\alpha, grd, upd)\!\!\rightarrow q$ and $p'\text{-}(\alpha', grd', upd')\!\!\rightarrow q'$, either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$. We implicitly assume a CA to be simple unless stated otherwise. CA constructed from regexes by the algorithm in [55] are simple.

Attainable memory. Given a CA A, its state $q \in Q$, and a counter memory $\mathfrak{m} \in \mathfrak{M}$, we say $\mathfrak{m}$ is *attainable* in q iff there exists a word w, such that some run of A on w visits the configuration $\langle q, \mathfrak{m}\rangle$. Put another way, $\mathfrak{m}$ is attainable in q iff $\mathcal{L}_L(\langle q, \mathfrak{m}\rangle) \neq \emptyset$ in $SFA(A)$. This means that the memory $\mathfrak{m}$ can actually exist in the state q during some (possibly non-accepting) run of A. Note that not all memories are attainable, and the sets of attainable memories differ between states. The set of all attainable memories of a state q is denoted $\mathfrak{M}(q) = \{\mathfrak{m} \in \mathfrak{M} \mid \exists w \in \Sigma^* \colon w \in \mathcal{L}_L(\langle q, \mathfrak{m}\rangle)\}$

Language of a CA state. Given a CA A and its state $q \in Q$, the *right language of* q – $\mathcal{L}_R(q) \subseteq \mathfrak{M} \times \Sigma^*$ – contains precisely those pairs of word w and memory $\mathfrak{m}$, for which w is accepted from q with $\mathfrak{m}$. Using $SFA(A)$, it is defined as $\mathcal{L}_R(q) = \{(\mathfrak{m}, w) \mid w \in \mathcal{L}_R(\langle q, \mathfrak{m}\rangle)\}$. The right language is also denoted $\mathcal{L}(q)$. The *left language of a CA state* is defined analogously: $\mathcal{L}_L(q) = \{(\mathfrak{m}, w) \mid w \in \mathcal{L}_L(\langle q, \mathfrak{m}\rangle)\}$.

Language via a state. Given a CA A and its state $q \in Q$, the *language via* q, denoted $\mathcal{L}_V(q)$, is the set of those words in $\mathcal{L}(A)$, for which an accepting run of A on w visits the state q. We define this set formally using the left and right language of q in $SFA(A)$: $\mathcal{L}_V(q) = \{w_L \cdot w_R \mid \exists\, \mathfrak{m} \in \mathfrak{M} \colon w_L \in \mathcal{L}_L(\langle q, \mathfrak{m}\rangle) \ \wedge \ w_R \in \mathcal{L}_R(\langle q, \mathfrak{m}\rangle)\}$.

Counter scope. In many CA encountered in practice, only relatively few of the states of the CA use counters. For example, it often happens that a counter is used in a single state, and in all other states, the value of this counter is zero (recall counters are initialized to zero in the beginning of CA run, and reset to zero on exit from a loop). We can take advantage of this by not explicitly storing the value of a counter in those states, where its value is always zero.

For this purpose, we define the *scope of counter* c, denoted S_c, as the set of those states in which c can have other value than zero. S_c is the smallest set of states such that

1. $p\text{-}(\alpha, upd, grd)\!\!\rightarrow q \ \wedge \ upd(c) \in \{\textsc{incr}, \textsc{exit1}\} \implies q \in S_c$,
2. $p\text{-}(\alpha, upd, grd)\!\!\rightarrow q \ \wedge \ p \in S_c \wedge upd(c) = \textsc{noop} \implies q \in S_c$.

In other words, the scope of c begins after it is incremented or reset to 1, and extends as far as c is not changed. For a state q and counter c, such that $q \notin S_c$, we say, perhaps somewhat illogically, that c is *out of scope* in q. We take this commonly used phrase from the jargon of programming languages, where one speaks of out-of-scope variables, and use it in the context of counters.

For a state q we define the *set of non-zero counters*, N_q, as the set of counters which are in scope in q (and thus can have non-zero value): $N_q = \{c \in C \mid q \in S_c\}$.

Example 2.4.3. In Fig. 2.1, the scope of c is $\{q\}$, as its value is always zero in p.

Chapter 3

Simulation in CA

In this chapter we present our main contribution. We first define the concepts of counter liveness and counter mapping. Using on these, we give the definition of simulation in CA parameterized by counter mapping, along with a proof sketch of its correctness with respect to language inclusion. We further describe some of its properties which help us grasp this novel concept.

3.1 Counter Liveness

This section presents the concept of liveness of a counter. It is a straightforward adaptation of the liveness of a variable, known from conventional static program analysis.

Definition 3.1.1 (**Counter liveness**)**.** A counter $c \in C$ is said to be *live in state* $q \in Q$ iff the value of c is read in some state r reachable from q, and there exists at least one path from q to r which does not reset c. Intuitively, liveness of a counter indicates that the counter value may be required at a later point, and needs to be stored in memory.

In the formal definition of liveness, we define sets of states for a counter c, each having certain property with respect to c. In order to do so, we introduce the following auxiliary definition. Given a sequence or set of transitions, $\beta \in \Delta^*$ or $\beta \subseteq \Delta$ respectively, let $Q[\beta]$ be the set of states occurring in β, defined as $Q[\beta] = \{q \in Q \mid p \text{-}(*)\!\!\rightarrow q \in \beta \vee q \text{-}(*)\!\!\rightarrow r \in \beta\}$.

For a counter c, we define the following sets of states:

- The set of initializing states of counter c:

$$I_c = \{q_0\} \cup \{\, i \in Q \mid q \text{-}(\alpha, grd, upd)\!\!\rightarrow i \wedge upd(c) \in \{\textsc{exit}, \textsc{exit1}\} \,\}. \tag{3.1}$$

  These are the states in which c has been reset on an incoming transition, and thus has value either 0 or 1 in this state.[8] This set includes the initial state q_0, since in the initial state all counters are implicitly initialized to zero.

- The set of utilizing states of counter c:

$$U_c = \{\, u \in Q \mid fin_u(c) \notin \{\textbf{true}, \textbf{false}\} \quad \vee \quad u \text{-}(\alpha, grd, upd)\!\!\rightarrow q \wedge upd(c) \neq \textsc{noop} \,\}. \tag{3.2}$$

[8] Note that in the state i, c may possibly acquire other values besides 0 or 1 if another transition leads to i, which performs another update (e.g. INCR or NOOP). If we are to be precise, we should say that i *may* have value either 0 or 1 in i. The concrete values are not important however; the important fact is that c is reset on some transition into i.

These are the states in which c is read; that is, in which c either occurs in the acceptance condition, or is updated on some outgoing transition by a non-NOOP update. We can safely ignore the NOOP update and its guard **true**, since neither of them requires the actual value of c for their evaluation – **true** is constant, and NOOP performs no action.

- The set of resetting transitions of counter c:

$$R_c = \{p \xrightarrow{(\alpha, grd, upd)} q \mid upd(c) \in \{\text{EXIT}, \text{EXIT1}\}\}. \tag{3.3}$$

These are the transitions, on which c is reset, either to 0 or 1, and thus its previous value is discarded.

- The set of live paths of counter c:

$$P_c = \{(i, \beta, u) \in (I_c \times \Delta^* \times U_c) \mid i \overset{\beta}{\rightsquigarrow} u \wedge \beta \cap R_c = \emptyset\}. \tag{3.4}$$

These are triples (i, β, u), where i is an initializing state for counter c; u is a utilizing state for c; and β is a sequence of transitions leading from i to u, such that no transition in β resets c. Note that β can be empty, in which case $i = u$ (this happens e.g. if c is utilized in the initial state). Intuitively, these are the paths along which c is live, since on each transition along the path, the new value of c depends on the previous one, and it will eventually be used.[9]

- **The set of states in which counter c is live**:

$$L_c = \{q \in Q \mid \exists\, (i, \beta, u) \in P_c \colon q \in \{i, u\} \cup Q[\beta]\}. \tag{3.5}$$

These are simply all the states along each live path of c. Intuitively, these are the states in which the value of c is still needed, since it may be read at some point in the future, while not being reset prior to being read. If it were reset before being read, we do not need to keep its value and can simply discard it immediately, since it would be discarded by the inevitable reset anyway.

The set of live counters in state q is defined accordingly:

$$C_q = \{c \in C \mid q \in L_c\}. \tag{3.6}$$

3.2 Counter Mapping

For our parameterized simulation in CA, we need to describe how the counters of weaker and stronger state are related. For this purpose, we use the concept of counter mapping, introduced in this section. Simply put, it determines which counter in the stronger state simulates which counter in the weaker state.

[9] To be exact, c will be used only if the sequence of transitions β is executable. Thus, the set U_c generally overapproximates the actual set of states in which c can be used. However, the derivative construction described in [55] does not produce CA with non-executable transitions, according to footnote 5 in [55].

Definition 3.2.1 (Counter mapping). A *counter mapping* $\gamma_q^s \colon C_s \to C_q$ assigns to each counter d live in the simulating state s a unique corresponding counter c live in the simulated state q. In other words, γ_q^s is total and injective, and is empty if $C_s = \emptyset$. For a counter c, we say c is *mapped via* γ_q^s (or simply *mapped* if clear from context) iff $c \in \mathbf{im}\,\gamma_q^s$; otherwise c is not mapped via γ_q^s. Since γ_q^s is injective, we freely interchange the direction of mapping and say c *is mapped to* d as well as d *is mapped to* c. We commonly omit the states and use just γ whenever clear from context; and we use the convention $c \equiv \gamma(d)$ unless specified otherwise.

The *identity mapping* γ_{ID} maps each counter to itself: $\forall c \in C \colon \gamma_{\text{ID}}(c) = c$. If used in a state q, we implicitly assume its domain and image to be restricted to C_q. As each counter mapping γ is injective, it has an *inverse*, denoted γ^{-1}.

We lift the definition to counter memories in the following way:

$$\gamma(\mathfrak{m}) = \{c \mapsto \mathfrak{m}(\gamma^{-1}(c)) \mid c \in \mathbf{im}\,\gamma\} \cup \{c \mapsto 0 \mid c \notin \mathbf{im}\,\gamma\}.$$[10]

That is, counters which are mapped via γ are renamed; those which are not, have value 0. Such memory $\gamma(\mathfrak{m})$ is said to be *remapped*. Counters not mapped from s via γ have the value 0, but their value does not matter in most contexts. Such counters are dead in s and thus their value will never be needed in s, whereas in q their value is, generally speaking, not relevant (we will address this issue later in Chapter 4). Note that in such remapped memory, the value of a counter may hypothetically exceed its upper bound. This is not a concern, as it will never happen in practice.

Totality and injectivity. We require that each counter d live in the simulating state s has precisely one *unique* corresponding counter c live in the simulated state q:

$$\text{Every counter mapping } \gamma_q^s \colon C_s \to C_q \text{ is total and injective.} \tag{3.7}$$

The reason behind this requirement is that a counter restricts the language of a state. Specifically, the language of a state is in general made smaller by adding a counter and larger by removing a counter. This is because counter guards may disable some transitions and limit the conditions under the automaton accepts in the given state. In order to ensure that q only accepts the words s accepts, we must apply the same restrictions on q which are present in s. We overapproximate these restrictions by requiring that for each live counter d in s, there must be a corresponding live counter c in q. Further, we require that there is no more than one such c for each d (γ is a function, not an arbitrary relation), and no two d, d' in s can correspond to the same c in q (the mapping is injective).

Example 3.2.1. Figure 3.1 shows a case when the totality requirement is too strict. The reason for the injectivity requirement is illustrated in Figures fig. 3.2a and 3.2b. Albeit neither of the examples presented therein arises in our CA constructed from regular expressions [55; 54], there may be a different case which does occur in practice.

In fig. 3.2a, multiple counters (c, d) in the weaker state q are simulated by a single counter (f) in the stronger state s. If the initial memory is the zero memory $\mathfrak{m}_0$, then s and q have the following languages respectively (recall $n \sim c$ means n is within the bounds of c):

$$\mathcal{L}(\langle s, \mathfrak{m}_0 \rangle) = \{w \in \Sigma^* \mid |w|_x + |w|_y \sim f\},$$
$$\mathcal{L}(\langle q, \mathfrak{m}_0 \rangle) = \{w \in \Sigma^* \mid |w|_x \sim c \wedge |w|_y \sim d\}.$$

[10] Let $\gamma \colon \mathfrak{M} \to \mathfrak{M}$ be a mapping lifted to memories. By its inverse, denoted γ^{-1}, we understand not the functional inverse of the lifted mapping γ, but the inverse of the base mapping, lifted to counter memories: $\gamma^{-1}(\mathfrak{m}) = \{c \mapsto \mathfrak{m}(\gamma(c)) \mid c \in \mathbf{im}\,\gamma^{-1}\} \cup \{c \mapsto 0 \mid c \notin \mathbf{im}\,\gamma^{-1}\}$.

Assume $\boldsymbol{max}_c = \boldsymbol{max}_d = \boldsymbol{max}_f = k$. Then for $w \in \Sigma^*$, such that $|w|_x = |w|_y = k$, we have $w \in \mathcal{L}(\langle q, \emptyset \rangle)$ but $w \notin \mathcal{L}(\langle s, \emptyset \rangle)$. Thus $\mathcal{L}(\langle q, \mathfrak{m}_\mathfrak{o} \rangle) \not\subseteq \mathcal{L}(\langle s, \mathfrak{m}_\mathfrak{o} \rangle)$.

In fig. 3.2b, the principle is similar. Single counter (c) in the weaker state q is simulated by multiple counters (f, g) in the stronger state s. With zero memory $\mathfrak{m}_\mathfrak{o}$, s and q have the following languages respectively:

$$\mathcal{L}(\langle s, \mathfrak{m}_\mathfrak{o} \rangle) = \{w \in \Sigma^* \mid |w|_a \sim f \ \wedge \ |w|_b \sim g\},$$
$$\mathcal{L}(\langle q, \mathfrak{m}_\mathfrak{o} \rangle) = \{w \in \Sigma^* \mid |w|_a + |w|_b \sim c\}.$$

Assume $\boldsymbol{min}_c = \boldsymbol{min}_f = \boldsymbol{min}_g = k > 0$. Then for $w \in \Sigma^*$, s.t. $|w|_a + |w|_b = k$, we have $w \in \mathcal{L}(\langle q, \emptyset \rangle)$ but $w \notin \mathcal{L}(\langle s, \emptyset \rangle)$. Thus $\mathcal{L}(\langle q, \mathfrak{m}_\mathfrak{o} \rangle) \not\subseteq \mathcal{L}(\langle s, \mathfrak{m}_\mathfrak{o} \rangle)$.

Example 3.2.2. Condition 3.7 is overly strict, as illustrated in Figures 3.2c and 3.2d. In the automata shown therein, multiple counters can simulate a single one, or vice versa, while maintaining language inclusion. Although in such cases the Condition 3.7 is too prohibitive, not including it would make the consequent reasoning about CA simulation more difficult. Moreover, we expect that in practical use-cases it will not induce a significant penalty in terms of CA size reduction.

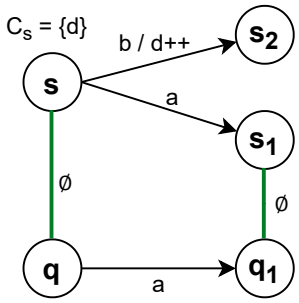
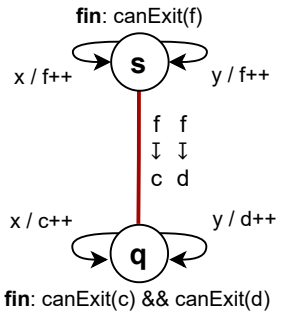
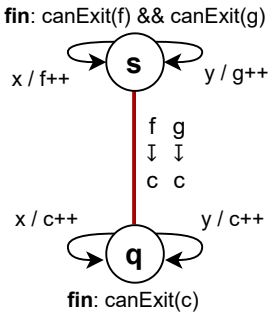


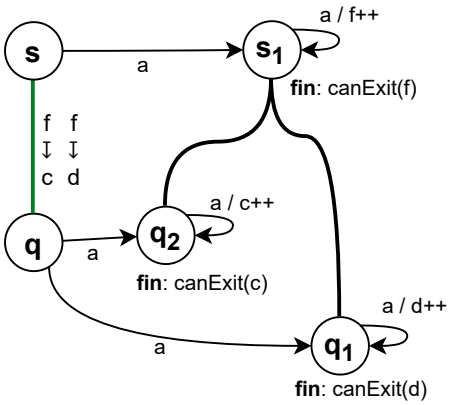
Figure 3.1: Stronger state s has a live counter d, but this counter does not need to be mapped in order for s to simulate q.



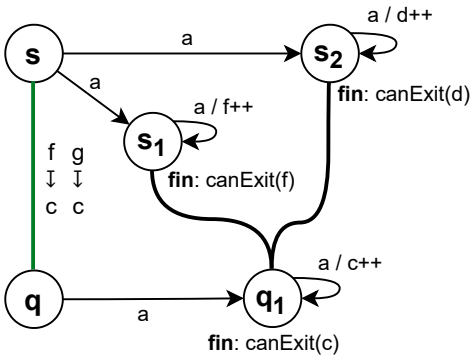
(a) Invalid mapping in which multiple counters are simulated by a single one.



(b) Invalid mapping in which one counter is simulated by multiple ones.



(c) Valid mapping in which multiple counters are simulated by a single one.



(d) Valid mapping in which one counter is simulated by multiple ones.

Figure 3.2: Counter mappings which simulate several counters by a single one, or vice versa.

3.3 Definition of CA Simulation

Before giving a formal definition of simulation in counting automata, we first build an intuition which will help us understand the different aspects of simulation in CA. Let us first recall simulation in classical finite automata. Its key property is that it implies language inclusion. That is, if $p \preccurlyeq q$, then $\mathcal{L}(p) \subseteq \mathcal{L}(q)$. As a result, if p and q are $\preccurlyeq$-*equivalent*, we can merge them while preserving language of the FA. In CA, we would like to do the same. However, the language of a CA state is *parametric* – it depends on the specific *counter memory* which the CA has in that state in a given moment during a run. We do not know the specific memory (or memories) when merging two states, as it is only known during the CA run.[11] Therefore, we have to require that the language of the weaker state is a subset of the language of the stronger state with *any* memory, as will be explained later. This leads us to the following informal definition of CA simulation (a formal definition is presented further below).

CA simulation defined informally. State s simulates state q under counter mapping γ (denoted $q \sqsubseteq_\gamma s$) if, given a fixed counter memory $\mathfrak{m}_s$, state s can "do as much" with $\mathfrak{m}_s$, as q can do with the remapped memory $\gamma(\mathfrak{m}_s)$. In more detail:

1. Whenever the weaker state q accepts with some memory $\mathfrak{m}$, the stronger state s accepts with that same memory remapped according to the mapping: $\gamma^{-1}(\mathfrak{m})$. (Note that the mapping is directed from stronger to weaker state, so in order to transform a memory from weaker to stronger, we have to use the inverse mapping γ^{-1}.)

2. Each transition from q is "simulated" by a transition from s. This "simulation" has two aspects:

   (a) Transition executability, given by input predicates and counter guards. Predicates and guards on the weaker transition must imply respective predicates and guards on the stronger one.

   (b) Updates on transitions and simulation of successors. The respective updates on weaker and stronger transitions must result in such memories in the weaker and stronger successor states, that the stronger memory $\mathfrak{m}'_s = upd_{\tau_s}(\gamma^{-1}(\mathfrak{m}_q))$ "simulates" the weaker memory $\mathfrak{m}'_q = upd_{\tau_q}(\mathfrak{m}_q)$. This means that in the stronger successor configuration $\langle s', \mathfrak{m}'_s \rangle$, we must accept at least all those words which we accept in the weaker successor configuration $\langle q', \mathfrak{m}'_q \rangle$.

   We ensure this by requiring that s' simulates q' under some successor mapping γ'. But the simulation between q' and s' holds only for memories which are equal under γ'. We thus also require that the successor memories are equal under the successor mapping: $\mathfrak{m}'_s = \gamma^{-1}(\mathfrak{m}'_q)$. Note that the successor mapping γ' can differ from γ, as long as the successor memories are equal under it. Example 3.3.1 illustrates this case.

This definition of simulation is inspired by our goal to merge simulation-equivalent states. Assume states p and q are simulation-equivalent. Then given any *memory* $\mathfrak{m}$ and *word* w, either both states accept w with $\mathfrak{m}$, or both reject. More precisely, if γ_q^p is the mapping from p to q, then either the CA accepts w from both $\langle p, \mathfrak{m} \rangle$ and $\langle q, \gamma_q^p(\mathfrak{m}) \rangle$, or from

[11] Although we could compute the set of attainable memories for each state using static analysis, this approach would be more complicated and presumably not very beneficial.

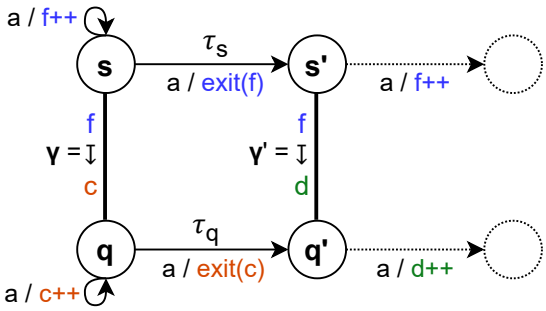


Figure 3.3: Counter mapping is changed on transitions to successors.

neither configuration. Based on this, we can merge the states p and q under the mapping γ_q^p (the mechanism of merging is described later, in chapter 4). The specific memories attainable in p and q are irrelevant, as the simulation equivalence holds for *all* counter memories.

Example 3.3.1. Fig. 3.3 illustrates a simulation where the mapping changes on transitions to successors. State s simulates state q under counter mapping γ, which maps counter f to c. Both these counters are reset on transitions τ_s and τ_q respectively. In the successor states s' and q', the mapping changed – f now maps to d. In the example, we assume d has zero value in q', allowing it to be mapped to f. Without this assumption, mapping f to d in γ' would not be possible.

Relation of simulation and language inclusion. We now formalize the informal statement given above: language of the weaker state q is a subset of the language of the stronger state s with any memory, under a counter mapping γ. More precisely, for any memory $\mathfrak{m}_q$, the language of q with $\mathfrak{m}_q$ is a subset of the language of s with $\gamma^{-1}(\mathfrak{m}_q)$:

$$q \sqsubseteq_\gamma s \implies \forall\, \mathfrak{m}_q \in \mathfrak{M}\colon \mathcal{L}(\langle q, \mathfrak{m}\rangle) \subseteq \mathcal{L}(\langle s, \gamma^{-1}(\mathfrak{m}_q)\rangle). \tag{3.8}$$

Relation to classical simulation in the configuration automaton. Observe that there is a connection between our parameterized simulation in a CA A, and classical simulation in its configuration automaton $SFA(A)$. Namely, if q is simulated by s under mapping γ, then every state $\langle q, \mathfrak{m}\rangle$ in $SFA(A)$ is simulated by $\langle s, \gamma^{-1}(\mathfrak{m})\rangle$:

$$q \sqsubseteq_\gamma s \implies \forall\, \mathfrak{m}_q \in \mathfrak{M}\colon \langle q, \mathfrak{m}_q\rangle \preccurlyeq \langle s, \gamma^{-1}(\mathfrak{m}_q)\rangle. \tag{3.9}$$

The converse of Eq. 3.9 does not hold, hence the simulation on the $SFA(A)$ induced by our simulation on A is an underapproximation of the actual simulation on $SFA(A)$. This is caused, among other limitations, by the following requirement: the respective operations on the weaker and stronger transition must either be equal, or can differ but only if they result in the same[12] successor memories, assuming the memories are the same[12] in the source states. Besides allowing different updates on weaker and stronger transitions, we also allow to use a *different mapping* between the successor states than between the current states, under certain conditions. The specific conditions are formalized in the following definition.

[12] Under the counter mapping, viz. $\mathfrak{m}_q = \gamma(\mathfrak{m}_s)$.

Definition 3.3.1 (**Simulation in counting automata**). A *simulation on a CA* A, $\sqsubseteq \subseteq Q \times Q \times (C_s \to C_q)$, is a relation assigning to each pair of states (q, s) of A at most one counter mapping between them, γ_q^s. We can thus think of $\sqsubseteq$ as a partial function $(Q \times Q) \rightharpoonup (C_s \to C_q)$. We use the notation $q \sqsubseteq_\gamma s \stackrel{\text{def}}{=} (q, s, \gamma) \in \sqsubseteq$. Due to the element γ which is not present in classical FA simulation, we say that CA simulation is *parameterized by counter mapping* between each stronger-weaker pair of states. Now follows the definition of CA simulation.

For states $q, s \in Q$, $q \sqsubseteq_\gamma s$ holds iff the following conditions hold:

(I) If q accepts, s must accept under the mapping γ:

$$q \in F \Rightarrow s \in F, \tag{3.10}$$
$$\forall\, d \in C_s \colon \mathit{fin}_q(\gamma(d)) \Rightarrow \mathit{fin}_s(d). \tag{3.11}$$

Note that in Condition 3.11, it suffices to consider live counters of s. This is because the acceptance condition of all dead counters is **true**, per definition of CA. Hence we can ignore them in $\mathfrak{fin}_s$, as it is a conjunction. If no counter is live in s, the Condition 3.11 implicitly holds.

(II) For each transition from q, $\tau_q \colon q \text{-}(\alpha, \mathit{grd}_{\tau_q}, \mathit{upd}_{\tau_q})\!\!\rightarrow\!\! q' \in \Delta$ there exists

(a) a *simulating transition* from s, $\tau_s \colon s \text{-}(\beta, \mathit{grd}_{\tau_s}, \mathit{upd}_{\tau_s})\!\!\rightarrow\!\! s' \in \Delta$, and
(b) a *successor counter mapping* $\gamma' = \gamma_{q'}^{s'}$ between the successor states q' and s',

such that all the following conditions hold (in which case we say that τ_s simulates τ_q under the (current) mapping γ and successor mapping γ'):

1. **Input predicate implication.**

$$\alpha \Rightarrow \beta. \tag{3.12}$$

2. **Counter guard implication.**

$$\forall d \in C_s \colon \mathit{grd}_{\tau_q}(\gamma(d)) \Rightarrow \mathit{grd}_{\tau_s}(d). \tag{3.13}$$

This is to ensure that the stronger transition τ_s is executable whenever the weaker transition τ_q is. By $\mathit{grd}_{\tau_q}(c) \Rightarrow \mathit{grd}_{\tau_s}(d)$, we mean that for every value $x \in \mathbb{N}$, if $\mathit{grd}_{\tau_q}(c)(x)$ holds, then $\mathit{grd}_{\tau_s}(d)(x)$ also holds.

3. **Simulation of successors.**

$$q' \sqsubseteq_{\gamma'} s'. \tag{3.14}$$

As in classical FA simulation, simulation must hold between successors. However, the mapping γ' between successors may differ from the current mapping γ.

4. **Counter update and successor counter mapping.**

For each $c \in C_q$ mapped via γ to some $d \in C_s$, either of the following holds for the successor mapping $\gamma' = \gamma_{s'}^{q'}$ between the successor states (q', s'):[13]

[13] These conditions are based on our particular case of CA constructed from regular expressions according to [55]. In different or more general cases, these conditions should be revised.

(a) **The mapping of c as well as d is dropped** – neither c nor d is mapped under γ'. This is to account for currently mapped counters that become dead in the successor states, so we no longer want to map them. The following conditions must hold when the mapping is to be dropped:

$$c \notin \mathbf{im}\,\gamma', \tag{3.15}$$
$$d \notin \mathbf{dom}\,\gamma'. \tag{3.16}$$

It follows from Cond. 3.16 that $upd_{\tau_s}(d) \in \{\text{NOOP}, \text{EXIT}, \text{EXIT1}\}$; otherwise d would be live in s' and thus would be mapped via γ'. Its guard is thus either $\top$ or CANEXIT_d. In the latter case, by Condition 3.13 $upd_{\tau_q}(c)$ is EXIT or EXIT1 – only guards of these updates can imply the guard CANEXIT_d. However, we do not care about the particular updates; it is sufficient that $grd_{\tau_q}(c)$ implies $grd_{\tau_s}(d)$.

(b) **The mapping is not dropped** – c is mapped to d' via γ', viz. $\gamma'(d') = c$, and one of the following Conditions A-F holds. The (presumably) most common case is A, in which the mapping stays the same. The remaining conditions address particular "corner cases". They allow us to obtain a larger simulation relation by changing the counter mapping on transitions to successors (as seen in Example 3.3.1).

(A) The mapping is the same as in the previous pair of states (q, s), and the updates on c and d' are equal:

$$d = d', \tag{3.17}$$
$$upd_{\tau_q}(c) = upd_{\tau_s}(d). \tag{3.18}$$

In this case, we say that the mapping of c is *preserved*.

(B) Both c and d' have been reset to the same value on τ_q and τ_s respectively:

$$upd_{\tau_q}(c) = upd_{\tau_s}(d') = u, \tag{3.19}$$
$$u \in \{\text{EXIT}, \text{EXIT1}\}. \tag{3.20}$$

Note that the mapping can be preserved also in this case: if the reset on c and d is the same, then this case coincides with Cond. A. If the mapping is not preserved, we say that c has been *remapped*.

The intuition behind this condition is that if both c and d' are reset on τ_q and τ_s respectively, then they have the same value after executing the respective transitions. Therefore, we can change the original mapping of c from d to d'. The motivation for such remapping is the case when we cannot simulate c by d in the successor state pair (q', s'), but we can simulate it by d'.

(C) c has been reset to 1 while d' has entered the scope by being incremented:

$$upd_{\tau_q}(c) = \text{EXIT1}, \tag{3.21}$$
$$d' \notin N_s \wedge upd_{\tau_s}(d') = \text{INCR}. \tag{3.22}$$

Again, the intuition is that if d' is out of scope on s, then after the increment on τ_s it has value 1 – same as c after its reset.

(D) c has entered the scope by being incremented while d' has been reset to 1:

$$c \notin N_q \wedge upd_{\tau_q}(c) = \text{INCR}, \tag{3.23}$$
$$upd_{\tau_s}(d') = \text{EXIT1}. \tag{3.24}$$

(E) c has been reset to 0 and d' is out of scope:

$$upd_{\tau_q}(c) = \text{EXIT}, \tag{3.25}$$
$$d' \notin N_{s'}. \tag{3.26}$$

Since d' is out of scope in s', its value in s' is always zero. Hence, as in previous conditions, c is assured to have the same value as d' after their respective transitions.

(F) c is out of scope and d' has been reset to 0:

$$c \notin N_{q'}, \tag{3.27}$$
$$upd_{\tau_s}(d') = \text{EXIT}. \tag{3.28}$$

Similarly as in the previous point, c will have the same value as d' after their respective transitions.

Validity of counter remapping. To see that the counter remapping is valid with respect to transition executability, observe the following.

1. In Condition B, d' is live in s, since it is reset on τ_s. Thus there is some $c' \in C_q$, s.t. $grd_{\tau_q}(c') \Rightarrow grd_{\tau_s}(d')$. Then on τ_q, we have a conjunction of counter guards $grd_{\tau_q}(c') \wedge grd_{\tau_q}(c) \wedge (\ldots)$. This conjunction implies $grd_{\tau_s}(d')$, since its first term – $grd_{\tau_q}(c')$ – implies it. Therefore, we can reset d' on τ_s whenever we can reset c on τ_q. More generally, we can perform τ_s whenever we can perform τ_q. The same principle applies to Conditions C and D. Furthermore, in Condition C, d' can be always incremented on τ_s, since it is out of scope in s.

2. In Condition E, as d' is out of scope in s', its update on τ_s must be either NOOP or EXIT. In the former case, its guard is always true and thus its update can be performed whenever we can reset c. In the latter case, d' must be live in s since it is reset on τ_s. Therefore, as in Conditions B-D, there exists some $c' \in C_q$, such that $grd_{\tau_q}(c') \Rightarrow grd_{\tau_s}(d')$. Due to this, we can perform τ_s whenever we can perform τ_q.

3. In Condition F, d' is live in s. Analogously to the previous conditions, there must exist some $c' \in C_q$, such that $grd_{\tau_q}(c')$ implies $grd_{\tau_s}(d')$. Due to this, we can perform τ_s whenever we can perform τ_q.

3.4 Correctness of Parameterized Simulation

This section presents the basic idea of a proof showing that the definition of parameterized simulation on CA is correct. Specifically this means that simulation on CA implies language inclusion, as stated in Equation 3.8. This property will be important in the simulation-based CA reduction presented in chapter 4.

Given a CA A, let $\tilde{A} = \mathit{SFA}(A)$. To show correctness of simulation on A, we need to prove Equation 3.8. If we recall Eq. 2.3, we can see that Eq. 3.9 implies 3.8. We thus adopt Equation 3.9 as our *induction hypothesis*, from which Equation 3.8 will follow.

Acceptance condition. We need to show that $\forall\, \mathfrak{m}_q \colon \langle q, \mathfrak{m}_q \rangle \in F_{\tilde{A}} \Rightarrow \langle s, \gamma^{-1}(\mathfrak{m}_q) \rangle \in F_{\tilde{A}}$. That is, whenever we can accept in q with some memory, we can remap this memory according to γ and accept with it in s. Recall that $\langle q, \mathfrak{m}_q \rangle \in F_{\tilde{A}} \iff q \in F \wedge \mathfrak{fin}_q(\mathfrak{m}_q)$. Therefore, we need to show $q \in F \wedge \mathfrak{fin}_q(\mathfrak{m}_q) \implies s \in F \wedge \mathfrak{fin}_s(\gamma^{-1}(\mathfrak{m}_q))$. Condition 3.10 states $q \in F \Rightarrow s \in F$. Now it remains to show

$$\mathfrak{fin}_q(\mathfrak{m}_q) \Rightarrow \mathfrak{fin}_s(\gamma^{-1}(\mathfrak{m}_q)). \tag{3.29}$$

Recall the acceptance condition $\mathfrak{fin}_s$ is a conjunction of individual acceptance conditions over all counters: $\mathfrak{fin}_s(\mathfrak{m}_s) \stackrel{\text{def}}{=} \bigwedge_{d \in C} \mathit{fin}_s(d)(\mathfrak{m}_s(d))$. Condition 3.11 states that the implication holds for each *mapped* counter, and for each mapped counter *separately*: $\forall\, c \in \mathbf{im}\,\gamma \colon \mathit{fin}_q(c) \Rightarrow \mathit{fin}_s(\gamma^{-1}(c))$. This, however, is sufficient for Eq. 3.29 to hold. Assume the antecedent holds, viz. $\forall\, c \in C \colon \mathit{fin}_q(c)(\mathfrak{m}_q(c))$. Then also $\mathit{fin}_s(\gamma^{-1}(c))(\mathfrak{m}_q(c))$ holds for each $c \in \mathbf{im}\,\gamma$, and so does their conjunction: $\bigwedge_{c \in \mathbf{im}\,\gamma} \mathit{fin}_s(\gamma^{-1}(c))(\mathfrak{m}_q(c))$. From this, we now obtain the desired result $\mathfrak{fin}_s(\gamma^{-1}(\mathfrak{m}_q))$.

We use the following fact which follows from the definition of counter mappings lifted to counter memories: $\forall\, d \in C_s \colon (\gamma^{-1}(\mathfrak{m}_q))(d) = (\mathfrak{m}_q \circ \gamma)(d)$. Here, γ is the ordinary counter mapping and γ^{-1} is its inverse lifted to counter memories.

$$\begin{aligned}
&\bigwedge_{c \in \mathbf{im}\,\gamma} \mathit{fin}_s(\gamma^{-1}(c))(\mathfrak{m}_q(c)) \\
&= \bigwedge_{c \in \mathbf{im}\,\gamma} \mathit{fin}_s(\gamma^{-1}(c))((\mathfrak{m}_q \circ \gamma \circ \gamma^{-1})(c)) \qquad (\text{since } c \in \mathbf{im}\,\gamma) \\
&= \bigwedge_{c \in \mathbf{im}\,\gamma} \mathit{fin}_s(\gamma^{-1}(c))((\mathfrak{m}_q \circ \gamma)(\gamma^{-1}(c))) \\
&= \bigwedge_{d \in \mathbf{dom}\,\gamma} \mathit{fin}_s(d)((\mathfrak{m}_q \circ \gamma)(d)) \\
&= \bigwedge_{d \in C_s} \mathit{fin}_s(d)((\mathfrak{m}_q \circ \gamma)(d)) \\
&= \bigwedge_{d \in C_s} \mathit{fin}_s(d)(\gamma^{-1}(\mathfrak{m})(d)) \\
&= \bigwedge_{d \in C} \begin{cases} \mathit{fin}_s(d)(\gamma^{-1}(\mathfrak{m})(d)) & \text{if } d \in C_s \\ \top & \text{otherwise} \end{cases}
\end{aligned}$$

Now recall that every counter $d \notin C_s$ has the acceptance condition $\mathit{fin}_S(d) = \mathbf{true}$, and its value in $\gamma^{-1}(\mathfrak{m})$ is 0. Therefore, $\mathit{fin}_S(d)(\gamma^{-1}(\mathfrak{m})(d)) = \mathbf{true}(0) = \top$ for all $c \notin C_s$. From this, we obtain

$$\begin{aligned}
&\bigwedge_{d \in C} \begin{cases} \mathit{fin}_S(d)(\gamma^{-1}(\mathfrak{m})(d)) & \text{if } d \in C_s \\ \mathit{fin}_S(d)(\gamma^{-1}(\mathfrak{m})(d)) & \text{otherwise} \end{cases} \\
&= \bigwedge_{d \in C} \mathit{fin}_S(d)(\gamma^{-1}(\mathfrak{m}_q)(d)) \\
&= \quad \mathfrak{fin}_S(\gamma^{-1}(\mathfrak{m}_q)).
\end{aligned}$$

Compatibility of transitions; simulation of successors. We need to show that if $q \sqsubseteq_\gamma s$ in A, then in $\tilde{A}$ for each $\mathfrak{m}_q$ and each transition $\tilde{\tau}_q \colon \langle q, \mathfrak{m}_q \rangle \text{-}\langle\alpha\rangle\!\!\rightarrow \langle q', \mathfrak{m}'_q \rangle$, there is a transition $\tilde{\tau}_s \colon \langle s, \mathfrak{m}_s \rangle \text{-}\langle\beta\rangle\!\!\rightarrow \langle s', \mathfrak{m}'_s \rangle$, such that

$$\mathfrak{m}_q = \gamma(\mathfrak{m}_s), \tag{3.30}$$

$$\alpha \Rightarrow \beta, \tag{3.31}$$

$$\langle q', \mathfrak{m}'_q \rangle \preccurlyeq \langle s', \mathfrak{m}'_s \rangle. \tag{3.32}$$

For a fixed transition τ_q in A from q to q' there is by Def. 3.3.1(II) a transition τ_s in A from s to s', which simulates τ_q for any memory pair $\mathfrak{m}_q$, $\mathfrak{m}_s = \gamma^{-1}(\mathfrak{m}_q)$. Specifically, Conditions 3.12 and 3.13 ensure that for any such memory pair, α implies β, and τ_s is executable whenever τ_q is (the latter can be proven similarly as we did with acceptance conditions). Thus there exists a $\tilde{\tau}_s$, for which Conditions 3.30 and 3.31 hold. Now the only thing left is to show that Condition 3.32 holds. The conditions specified in Def. 3.3.1(II)4 ensure that the successor memories $\mathfrak{m}'_q$ and $\mathfrak{m}'_s$ are equal under the successor mapping γ':

$$\mathfrak{m}'_q = \gamma'(\mathfrak{m}'_s). \tag{3.33}$$

By Condition 3.14 we have $q' \sqsubseteq_{\gamma'} s'$. Via the induction hypothesis 3.9, we then obtain $\mathfrak{m}'_q = \gamma'(\mathfrak{m}'_s) \implies \langle q', \mathfrak{m}'_q \rangle \preccurlyeq \langle s', \mathfrak{m}'_s \rangle$. The antecedent holds by 3.33, and thus the consequent – Condition 3.32 – holds also. We have thereby shown that Equation 3.9 holds, which implies that so does Equation 3.8.

3.5 Hypersimulation and its Properties

CA simulations are significantly more complex than classical FA simulations. For example, a CA can have several simulation preorders which may overlap. For this reason, we introduce the concept of *hypersimulation*, which is a generalization of CA simulation allowing multiple counter mappings between two states. It will allow us to effectively reason about all possible simulations of a CA. Following the definition, we then describe interesting closure properties of hypersimulations and simulations. We also introduce the concept of *consistent* simulation. This concept is important for simulation-based CA reduction, as a simulation which is not consistent generally cannot be used for reduction.

Definition 3.5.1 (**Hypersimulation on CA**)**.** A *hypersimulation on a CA A* is a union of arbitrarily many simulations on A. It is a relation $\boldsymbol{\mathcal{S}} \subseteq Q \times Q \times (C_s \to C_q)$, such that there exist n simulations $\sqsubseteq_{(i)}$; $1 \leq i \leq n$, for which $\mathcal{S} = \bigcup_{\forall i} \sqsubseteq_{(i)}$. Is assigns to each pair of states (q, s) a (possibly empty) set of counter mappings between them, S_q^s. We can thus think of $\mathcal{S}$ as a total function $(Q \times Q) \to \{C_s \to C_q\}$. We use the notation $q\, \mathcal{S}_\gamma\, s \stackrel{\text{def}}{=} (q, s, \gamma) \in \mathcal{S}$. By $|\mathcal{S}|$ we understand $\sum_{(p,q)\in Q\times Q} |\mathcal{S}(p, q)|$.

Definition 3.5.2 (**Ambiguity of hypersimulation**)**.** A hypersimulation $\mathcal{S}$ is *ambiguous* iff $\exists\, S_q^s \in \mathbf{im}\, \mathcal{S} \colon |S_q^s| > 1$. A hypersimulation which is not ambiguous is *unambiguous*.

The set of unambiguous hypersimulations of a CA A is trivially isomorphic to the set of simulations of A. The only difference is that a hypersimulation assigns to a pair of sates a *set of mappings*, whereas a simulation assigns a *single mapping*. We may thus use the terms "unambiguous hypersimulation" and "simulation" interchangeably whenever clear from context.

Theorem 3.5.1. The set of hypersimulations of a CA A is closed under union; the set of simulations of A is not.

Proof. Hypersimulations – immediate from the definition. Simulations – consider $\{(p, q, \gamma_1)\}$ and $\{(p, q, \gamma_2)\}$ where $\gamma_2 \neq \gamma_1$. Their union is ambiguous, hence not a simulation. □

Definition 3.5.3 (**Maximal and largest hypersimulation**)**.** A hypersimulation $\mathcal{S}$ on CA A is *maximal* iff it is not a subset of any other hypersimulation of A. $\mathcal{S}$ is *largest* iff there is no other hypersimulation $\mathcal{S}'$ on A, such that $|\mathcal{S}'| > |\mathcal{S}|$. As shown below, the maximal hypersimulation on A is unique; we denote it $\overline{\mathcal{S}}(A)$, or simply $\overline{\mathcal{S}}$ whenever clear from context.

Theorem 3.5.2. Among all hypersimulations of a CA, there is only one maximal hypersimulation, and thus only one largest hypersimulation.

Proof. Immediate from closure of hypersimulations under union. □

Having two separate terms *largest* and *maximal* seems pointless if they always agree. However, they will be used in restricted contexts, where we speak of maximal/largest hypersimulations of a certain kind (e.g. transitive, defined further below). In these restricted contexts, there may be several largest and maximal hypersimulations, and they may not agree.

Definition 3.5.4 (**Reflexivity**)**.** A hypersimulation $\mathcal{S}$ is *reflexive* iff $\forall q \in Q \colon q\ \ \mathcal{S}_{\gamma_{\text{ID}}}\ \ q$.

Definition 3.5.5 (**Reflexive closure**)**.** Given a hypersimulation $\mathcal{S}$, its *reflexive closure*, denoted $\mathcal{S}^{\text{R}}$, is its smallest superset which is a reflexive hypersimulation.

Lemma 3.5.1. The set of hypersimulations of a CA A is closed under reflexive closure; the set of simulations of A is not.

Proof. First case is obvious, as each state can simulate itself under the identity mapping. In the second case, the simulation $\sqsubseteq$ may contain a non-identity mapping for some (q, q); adding γ_{ID} for this pair will cause $\sqsubseteq$ to become ambiguous. □

Definition 3.5.6 (**Strong and weak symmetry**)**.** A hypersimulation $\mathcal{S}$ is *weakly symmetric* iff $\forall\, (p, q) \in Q \times Q \colon \mathcal{S}(p, q) \neq \emptyset \iff \mathcal{S}(q, p) \neq \emptyset$. In a hypersimulation $\mathcal{S}$, pair $(p, q) \in Q \times Q$ is *strongly symmetric* or simply *symmetric* iff $\mathcal{S}(p, q) = \{\gamma^{-1} \mid \gamma \in \mathcal{S}(q, p)\}$. $\mathcal{S}$ is *strongly symmetric* iff all $(p, q) \in Q \times Q$ are symmetric in $\mathcal{S}$.

Lemma 3.5.2. A hypersimulation has exactly one maximal weakly symmetric subset and one maximal strongly symmetric subset.

Proof. Assume a hypersimulation $\mathcal{S}$ has two maximal weakly symmetric subsets, $\mathcal{S}_1$ and $\mathcal{S}_2$. Their union clearly is a weakly symmetric subset of $\mathcal{S}$ strictly greater than either of $\mathcal{S}_1$, $\mathcal{S}_2$; hence, $\mathcal{S}_1$ and $\mathcal{S}_2$ are not maximal – contradiction. Strongly symmetric case is analogous. □

Note. Like classical FA simulations, CA (hyper)simulations are generally not closed under symmetric closure. Hence we purposefully do not define the symmetric closure of a CA (hyper)simulation, as it is of little practical utility.

Definition 3.5.7 (**Transitivity**)**.** A hypersimulation $\mathcal{S}$ is *transitive* iff $p\,\mathcal{S}_{\gamma_1}\,q \wedge q\,\mathcal{S}_{\gamma_2}\,r$ implies $p\,\mathcal{S}_{\gamma_3}\,r$, where $\gamma_3 = \gamma_1 \circ \gamma_2$.

Definition 3.5.8 (**Transitive fragment**)**.** Given a hypersimulation $\mathcal{S}$, its *transitive fragment* is any subset of $\mathcal{S}$ which is a transitive hypersimulation.

Lemma 3.5.3. The set of transitive hypersimulations is not closed under union. Therefore, a hypersimulation can (possibly) have multiple maximal transitive fragments.

Proof. The union of transitive hypersimulations $\{(p, q, \gamma_1)\}$ and $\{(q, r, \gamma_2)\}$ is not transitive. These are both maximal transitive fragments of their union. □

Definition 3.5.9 (**Transitive closure**)**.** Given a hypersimulation $\mathcal{S}$, its *transitive closure*, denoted $\mathcal{S}^{\mathrm{T}}$, is its smallest superset which is a transitive hypersimulation.

Theorem 3.5.3. The set of simulations of a CA A is not closed under transitive closure.

Proof. Consider a simulation $\{(p, q, \gamma_1), (q, r, \gamma_2), (p, r, \gamma_3)\}$, s.t. $\gamma_3 \neq \gamma_1 \circ \gamma_2$. Its transitive closure is ambiguous, as it contains both (p, r, γ_3) and $(p, r, \gamma_1 \circ \gamma_2)$.

Theorem 3.5.4. Let $\mathcal{S}'$ be a simulation where the counter mapping does not change across transitions, viz. $q \sqsubseteq_\gamma s \Rightarrow \forall q \text{-}(\ldots)\!\!\rightarrow q'\, \exists s \text{-}(\ldots)\!\!\rightarrow s' \colon q' \sqsubseteq_\gamma s'$. The set of such simulations is closed under transitive closure.

Proof. We proceed in two steps:

1. Each hypersimulation has a minimal transitive superset, which may not be a hypersimulation.

   We can construct this superset by a straightforward extension of any standard algorithm for computing transitive closure of a binary relation. This extension would, apart from considering the pairs of states (p, q), also consider the mapping between them. When adding a new element to the relation obtained from (p, q, γ_1) and (q, r, γ_1), this element will be $(p, r, \gamma_1 \circ \gamma_2)$. Observe that $\mathbf{dom}\,\gamma_1 \subseteq \mathbf{im}\,\gamma_2$, so such composition is well-defined, and is still injective.

2. The transitive superset thus computed is indeed a hypersimulation $\mathcal{S}^{\mathrm{T}}$.

   Assume $\mathcal{S}'$ where:

   (a) $\tau_q \colon q \text{-}(\ldots)\!\!\rightarrow q'$ is simulated by $\tau_r \colon r \text{-}(\ldots)\!\!\rightarrow r'$ under mapping γ_1 (which is both the current and successor mapping, as mappings do not change in $\mathcal{S}'$),

   (b) τ_r is simulated by $\tau_s \colon s \text{-}(\ldots)\!\!\rightarrow s'$ under γ_2 (likewise, both current and successor mapping).

To show that $(\mathcal{S}')^{\mathrm{T}}$ is indeed a simulation, we show the following: in $(\mathcal{S}')^{\mathrm{T}}$, τ_q can be simulated by τ_s under $\gamma_3 = \gamma_1 \circ \gamma_2$ as the current and successor mapping.

First, consider Conditions 3.10-3.13 in the definition of CA simulation. It is easily seen that if they all hold for both (1) τ_q and τ_r under γ_1, and (2) τ_r and τ_s under γ_2, then they hold for τ_q and τ_s under γ_3.

The only applicable rule for successor counter mapping is A. The updates are the same on d and $\gamma_2(d)$ for all $d \in C_s$, and in turn on $\gamma_2(d)$ and $\gamma_1(\gamma_2(d))$. Hence they are the same on d and $\gamma_1(\gamma_2(d))$. As a result, $\mathfrak{m}'_q = (\gamma_1 \circ \gamma_2)(\mathfrak{m}'_s)$, were $\mathfrak{m}'_q$ is the memory in q' after τ_q, and $\mathfrak{m}'_s$ is the memory in s' after τ_s. This means τ_s can simulate τ_q in $(\mathcal{S}')^{\mathrm{T}}$ *if* there is the mapping $\gamma_1 \circ \gamma_2$ from s' to q' in $(\mathcal{S}')^{\mathrm{T}}$. Now since the mappings do not change on transitions, $(q', r', \gamma_1) \in \mathcal{S}'$ and $(r', s', \gamma_2) \in \mathcal{S}'$ hold. Therefore, in the transitive closure $(\mathcal{S}')^{\mathrm{T}}$, there indeed is $\gamma_1 \circ \gamma_2$ from s' to q'.

□

Conjecture 3.5.1. The set of hypersimulations of a CA is closed under transitive closure.

Proof (idea). We may proceed similarly as in the special case above, where counter mappings do not change across transitions. We ought to to show that $\gamma'_3 = \gamma'_1 \circ \gamma'_2$ is a valid successor mapping for the simulation of τ_q by τ_s. A full proof would essentially require enumerating all the possible combinations of successor counter mappings (Conditions A-F) for the two pairs – τ_q simulated by τ_r, and τ_r simulated by τ_s. Each case may then be examined separately. This proof will also require further assertions about the underlying CA.[14] We do not have such full proof, as it would be very complex. However, precisely due to the complexity of the counter remapping conditions, a proof of their correctness (with regard to transitivity) is highly desirable.

We do not assume Conjecture 3.5.1 to hold, as we have no convincing proof of closure of hypersimulations under transitive closure.

Assumption 3.5.1. Conjecture 3.5.1 does not hold; that is, the set of hypersimulations of a CA A is *not* closed under transitive closure.

We will return to this assumption shortly, after we present the concept of *consistent simulation*. Essentially, a consistent simulation can be used for reduction of CA, whereas an inconsistent simulation cannot.

Definition 3.5.10 (Consistency). A hypersimulation is *consistent* iff it is unambiguous, reflexive and transitive. A hypersimulation which is not consistent is *inconsistent*.

Lemma 3.5.4. No ambiguous hypersimulation is consistent; unambiguous hypersimulation may or may not be consistent.

Intuitively, consistency of a simulation assures that it is "well-behaved" for the purposes of merging states of a CA. Namely, it does not require mapping a state to itself under a non-identity mapping, there is at most one mapping between two states, and its mappings all "agree" (by which we mean transitivity). Transitivity of a simulation would not be necessary for merging states, had hypersimulations been closed under transitive closure.

[14] One such assertion required for a full proof is that counters are used only in loops – if a counter is in scope, it is in a counting loop in which it is incremented. This property holds in our particular case.

However, by Assumption 3.5.1 this is not the case, and therefore we require that a consistent simulation is transitive.

We will concern ourselves only with computation and utilization of consistent CA simulations. This restriction to consistent simulations is inspired by our primary application of simulation in CA reduction. We want to merge states which are simulation-equivalent under a counter mapping. When merging three or more states, we need the respective mappings to "agree"; otherwise it would be very difficult to reason about the correctness of such merging. This is also the reason why we require that in a reflexive simulation, the mapping between each state and itself is the identity mapping. Allowing a non-identity mapping would only make matters more complicated and bring no benefit, since it makes no sense to merge a state with itself.

Lemma 3.5.5. In a consistent simulation $\sqsubseteq$, there is at most one mapping between each state and itself, and it is the identity mapping γ_{ID}.

Proof. Immediate from $\sqsubseteq$ being unambiguous and reflexive. □

Lemma 3.5.6. In a consistent simulation, if there is *any* $(p, q) \in Q \times Q$, such that there is a mapping from p to q and from q to p, then these mappings are the inverse of each other.[15]

Proof. Let $\gamma_1 = \sqsubseteq (q, p)$ and $\gamma_2 = \sqsubseteq (p, q)$. If the mapping γ_1 was not equal to $(\gamma_2)^{-1}$, then $\gamma_1 \circ \gamma_2 \neq \gamma_{\text{ID}}$. By transitivity, $\sqsubseteq (q, q) = \gamma_1 \circ \gamma_2 \neq \gamma_{\text{ID}}$, which contradicts $\sqsubseteq$ being reflexive. Thus $\sqsubseteq (p, q)$ must be equal to $\sqsubseteq (q, p)^{-1}$. □

Corollary 3.5.1. In a consistent simulation, weak and strong symmetry are equivalent.

Definition 3.5.11 (Simulation preorder). A consistent simulation $\sqsubseteq$ on CA A is a *maximal consistent simulation* iff it is maximal among consistent simulations on A. Analogously is defined a *largest consistent simulation*. We refer to a maximal consistent simulation as a *simulation preorder*.[16] We do not consider any inconsistent simulation to be a simulation preorder.

Theorem 3.5.5. Any simulation preorder is indeed a preorder relation; that is, reflexive and transitive under the definition of these terms for CA simulations.

Proof. Immediate from the definition of simulation preorder and consistency. □

Non-uniqueness of simulation preorders. Non-closure of simulations under union hints that there generally does not exist a single simulation preorder for a given CA A. This is indeed the case, and there are two reasons. First, there can be multiple counter mappings between a single pair of states, meaning the maximal hypersimulation $\overline{\mathcal{S}}$ is ambiguous. Second, the maximal hypersimulation $\overline{\mathcal{S}}$ may not be transitive.

Theorem 3.5.6. A counting automaton can have multiple simulation preorders. This holds also if its maximal hypersimulation $\overline{\mathcal{S}}$ is unambiguous.

[15] Observe that this statement is not equivalent with *"In a consistent simulation, weak symmetry implies strong symmetry"*; it is strictly stronger.

[16] Noting, as with classical FA simulation, that not all consistent simulation which are preorder relations are also maximal. We use the term *simulation preorder* to refer only to the maximal ones.

Proof. Consider the following $\overline{\mathcal{S}}$ on a 2-state, 2-counter CA A:

$$\{(p,q,\{c\mapsto c,\ d\mapsto d\}),\ (p,q,\{c\mapsto d,\ d\mapsto c\}),\ (p,p,\gamma_{\text{ID}}),\ (q,q,\gamma_{\text{ID}})\}.$$

It contains the following two simulation preorders:

$$\{(p,q,\{c\mapsto c,\ d\mapsto d\}),\ (p,p,\gamma_{\text{ID}}),\ (q,q,\gamma_{\text{ID}})\},$$
$$\{(p,q,\{c\mapsto d,\ d\mapsto c\}),\ (p,p,\gamma_{\text{ID}}),\ (q,q,\gamma_{\text{ID}})\}.$$

Now consider the following *unambiguous* $\overline{\mathcal{S}}$:

$$\{(p,q,\gamma_1),\ (q,r,\gamma_2),\ (p,p,\gamma_{\text{ID}}),\ (q,q,\gamma_{\text{ID}}),\ (r,r,\gamma_{\text{ID}})\}.$$

It contains the following two simulation preorders:

$$\{(p,q,\gamma_1),\ (p,p,\gamma_{\text{ID}}),\ (q,q,\gamma_{\text{ID}}),\ (r,r,\gamma_{\text{ID}})\},$$
$$\{(q,r,\gamma_2),\ (p,p,\gamma_{\text{ID}}),\ (q,q,\gamma_{\text{ID}}),\ (r,r,\gamma_{\text{ID}})\}.$$

□

Using several simulation preorders. If possible, we would like to reduce a CA by several simulation preorders, to obtain a better reduction. For example, we could merge states according to the *union* of some or all simulation preorders. However, as the following theorem tells us, that this is not possible.

Theorem 3.5.7. Given a CA A, the union of multiple simulation preorders of A is inconsistent.

Proof. Consider a CA A with $k > 1$ distinct simulation preorders $\sqsubseteq_{(i)}; 1 \le i \le k$. Assume their union $\sqsubseteq'$ is consistent. Then $\sqsubseteq'$ is a simulation preorder, and also a proper superset of $\sqsubseteq_{(j)}$ for some j. Thus this $\sqsubseteq_{(j)}$ is not maximal among consistent simulations; hence not a simulation preorder. This contradicts the assumption that $\sqsubseteq'$ is consistent. □

Simulation preorder in special cases. We now describe the properties of simulation preorders under some special cases of $\overline{\mathcal{S}}$. These special cases are interesting because we expect them to often arise in practice.

Theorem 3.5.8. If the maximal hypersimulation $\overline{\mathcal{S}}$ of a CA A is unambiguous and transitive, A has a single simulation preorder.

Proof. The simulation preorder is $\overline{\mathcal{S}}$, since it is reflexive, transitive, and unambiguous; thereby consistent. □

We expect especially this first case, addressed in Theorem 3.5.8, to be often encountered in practice. The following case will probably be less frequent; we include it nonetheless for completeness.

Lemma 3.5.7. In a CA A with unambiguous $\overline{\mathcal{S}}$, the set of simulation preorders coincides with the set of maximal transitive fragments *(MTF)* of $\overline{\mathcal{S}}$.[17]

[17] Where *maximal transitive fragment* is interpreted according to Def. 3.5.8.

Proof. MTF $\Rightarrow$ simulation preorder: $\overline{\mathcal{S}}$ is reflexive; thus each its MTF $\sqsubseteq$ is reflexive, since reflexive pairs (e.g. $q \sqsubseteq_{\gamma_{\mathrm{ID}}} q$) do not affect transitivity. As $\sqsubseteq$ is also transitive and unambiguous, it is consistent. Since consistency implies transitivity, it must be a maximal *consistent* fragment; hence a simulation preorder. If it was not, then there would be a greater consistent fragment than $\sqsubseteq$, therefore a greater transitive fragment, contradicting the premise that $\sqsubseteq$ is a maximal transitive fragment.

Simulation preorder $\Rightarrow$ MTF: Assume a simulation preorder $\sqsubseteq$ is not a MTF. Then there exists a MTF $\sqsubseteq'$ which is a proper superset of $\sqsubseteq$. This MTF $\sqsubseteq'$ is consistent, since each MTF is consistent (as shown above). This means $\sqsubseteq'$ is a greater consistent simulation than $\sqsubseteq$, hence $\sqsubseteq$ is not a simulation preorder – contradiction. □

(Bi)simulation equivalences. We now define the concepts of *bisimulation* and *simulation equivalence* in CA.

Definition 3.5.12 (**Equivalence class**)**.** Given a consistent simulation $\sqsubseteq$ on a CA A and a set of states $R \subseteq Q$, R is an *equivalence class* of $\sqsubseteq$ iff $\forall\, p, q \in R\colon \exists\, \gamma\colon p \sqsubseteq_\gamma q$. In other words, every two states p and q in R $\sqsubseteq$-simulate each other. It follows from $\sqsubseteq$ being consistent that the two mappings γ_p^q and γ_q^p, under which they simulate each other, are inverse of each other, viz. $\gamma_p^q = (\gamma_q^p)^{-1}$.

Definition 3.5.13 (**Bisimulation on CA**)**.** A *bisimulation* on a CA A is any strongly symmetric consistent simulation on A.

Definition 3.5.14 (**Hyperbisimulation on CA**)**.** A *hyperbisimulation on a CA* A is a union of arbitrarily many bisimulations on A. The *maximal hyperbisimulation* is the union of all hyperbisimulations.

Definition 3.5.15 (**Bisimulation equivalence**)**.** A *bisimulation equivalence* on a CA A is any bisimulation which is maximal among bisimulations on A. We denote a bisimulation equivalence by $\div$.

Definition 3.5.16 (**Simulation equivalence**)**.** A *simulation equivalence* on a CA A is a maximal strongly symmetric subset of any consistent simulation on A. We denote a simulation equivalence by $\doteq$. By $Q/\doteq$ we understand the set of equivalence classes of Q with respect to $\{(p, q) \mid \exists\, \gamma\colon (p, q, \gamma) \in \doteq\}$. We may occasionally refer by this term to a non-maximal equivalence if clear from the context.

Definition 3.5.17 (**Optimal equivalence**)**.** A simulation equivalence $\doteq$ on a CA A is *optimal* (with respect to A) iff there exists no other simulation equivalence $\doteq'$ on A, such that $|Q/\doteq'| < |Q/\doteq|$. Optimal bisimulation equivalence is defined analogously.

Intuitively, a simulation equivalence $\doteq$ is optimal iff it partitions Q into as few equivalence classes as any other simulation equivalence on A, or fewer. This means that after merging states according to $\doteq$, the number of states of the reduced automaton is as low as with any other simulation equivalence, or lower.

Lemma 3.5.8. Every optimal simulation equivalence is maximal among simulation equivalences, but not vice versa. Not every optimal simulation equivalence is largest; the converse case does not hold either. These statements also hold for bisimulation equivalences.

Proof. We only show a proof for simulation equivalences. For bisimulation equivalences, the proof is analogous.

Optimal $\Rightarrow$ *maximal*: assume an optimal equivalence $\doteq$ is not maximal. Then it is a proper subset of some other equivalence, which relates two classes which are not related in $\doteq$. Therefore, $\doteq$ cannot be optimal – contradiction.

Maximal $\not\Rightarrow$ *optimal*: consider the following $\overline{\mathcal{S}}$ (using simplified notation with counter mappings omitted):

$$\{\{p,q\},\{q,r\},\{r,s\},\{s,q\}\}.$$

Assume it has the maximal simulation equivalences $\{\{p,q\}\}$ and $\{\{q,r\},\{r,s\},\{s,q\}\}$. Only the second one is optimal.

Largest: consider the largest simulation equivalence (again in simplified notation):

$$\{a=b=c=d=e\}.$$

It has 25 elements – $\{(a,a),(a,b),\ldots(b,a),(b,b),\ldots\}$; and allows merging 5 states into one, hence eliminating 4 states. On the other hand, the equivalence

$$\{a=f,\ b=g,\ c=h,\ d=i,\ e=j\}$$

has only 20 elements, but allows eliminating 5 states. □

Searching for simulation preorders and equivalences. Assume we have $\overline{\mathcal{S}}$ computed and wish to obtain from it simulation preorders and subsequently simulation equivalences, which will be used for state merging. Because a CA can have multiple simulation preorders, we need to *search* for them in $\overline{\mathcal{S}}$, at least in the general case. For an ambiguous $\overline{\mathcal{S}}$, a basic naïve approach would be to examine each maximal transitive fragment of $\overline{\mathcal{S}}$ and search for simulation preorders within it (as these preorders are all transitive and thus must be contained within a maximal transitive fragment). But since a simulation preorder must itself be transitive, this approach requires to search for transitive fragments (i.e., the simulation preorders) *within* a transitive fragment. This probably wouldn't be more efficient than searching for simulation preorders directly in $\overline{\mathcal{S}}$.

In examples occurring in practice,[18] the number of possible mappings between two states in the maximal hypersimulation almost never exceeds 1. In this case, $\overline{\mathcal{S}}$ is unambiguous and we can employ a simpler naïve approach. This simpler approach would merely enumerate all maximal transitive fragments of $\overline{\mathcal{S}}$, without a need to choose from multiple possible counter mappings between two states. This will give us all simulation preorders in $\overline{\mathcal{S}}$, per Lemma 3.5.7. However, enumerating all maximal transitive fragments is not particularly easier than searching for all simulation preorders within an ambiguous $\overline{\mathcal{S}}$. This is because we cannot simply search for any maximal transitive subset; this subset has to be a *simulation*.

In Section 5.1 we present Algorithm Pseudosim, which for a given CA computes its maximal hypersimulation $\overline{\mathcal{S}}$. More precisely, it computes a certain *superset* of $\overline{\mathcal{S}}$ in general. This is caused by the ambiguity of $\overline{\mathcal{S}}$, due to which the algorithm may not be able to compute the actual $\overline{\mathcal{S}}$ but only its overapproximation.

Section 5.2 then presents Algorithm Search. This algorithm takes the superset of $\overline{\mathcal{S}}$ computed by Algorithm Pseudosim, and searches within it for consistent simulations (including those which are simulation preorders).

[18] Specifically in the CA constructed from regular expressions, via the construction presented in [55], as implemented in [54].

Chapter 4

Simulation-Based Reduction of CA

In this chapter, we present a method of reducing the size of a counting automaton based on the parameterized simulation defined in section 3.3. This method operates by merging states which are simulation-equivalent under a counter mapping. If we have a bisimulation (or bisimulation equivalence) $\fallingdotseq$, we can use it directly for merging, as it is always an equivalence. For a simulation, we have to reduce it into a simulation equivalence.

Obtaining simulation equivalence from a simulation. Assume we have a consistent simulation $\sqsubseteq$. We obtain a simulation equivalence $\doteq$ from it by taking its *maximal strongly-symmetric subset*. That means removing all $p \sqsubseteq_\gamma q$, such that $q \sqsubseteq_{(\gamma^{-1})} p$ does not hold. Formally, $\doteq \; = \{ (p, q, \gamma) \in \sqsubseteq \mid (q, p, \gamma^{-1}) \in \sqsubseteq \}$.

Merging method. We now wish to merge several states, which are all equivalent under the equivalence relation (either $\doteq$ or $\fallingdotseq$). The method selects one of the equivalent states as the *representative* of the equivalence class. The representative state stays unchanged, and the other states are merged into it. This merging consists of redirecting incoming and outgoing transitions of the merged state to go into / from the representative. On redirected incoming transitions, we much change from the "context" of the merged state to the "context" of the representative, and on outgoing transitions the other way. By changing context we mean modifying the counter memory (retained during a CA run) according to the mapping. Namely, if γ is a mapping from the representative to the merged state, then on incoming transitions we must change the CA memory from $\mathfrak{m}$ to $\gamma^{-1}(\mathfrak{m})$, and on outgoing from $\gamma^{-1}(\mathfrak{m})$ back to $\mathfrak{m} = \gamma(\gamma^{-1}(\mathfrak{m}))$. To allow such dynamic remapping of counter memory during a CA run, we introduce a *counter rename* operation on transitions. For this purpose, we introduce a slightly modified version of CA, named *renaming CA* (RCA), which allows renaming of counter on transitions according to a counter mapping.

Definition 4.0.1 (Renaming CA (RCA)). A *renaming counting automaton (RCA)* is a tuple $\widehat{A} = (\mathbb{I}, C, Q, q_0, F, \mathit{fin}, \Delta)$, where all elements except Δ are defined equivalently as in counting automata (Def. 2.4.1). The transition relation Δ is modified thus (added elements underlined):

$$\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times \underline{(\mathfrak{M} \to \mathfrak{M})} \times (C \to \textsc{Upd}) \times (C \to \textsc{Grd}_C) \times \underline{(\mathfrak{M} \to \mathfrak{M})} \times Q,$$

$$\tau \colon p \text{-}\!\!\langle \alpha, \underline{\gamma_1}, \mathit{upd}_\tau, \mathit{grd}_\tau, \underline{\gamma_2} \rangle\!\!\to q \in \Delta.$$

The newly added elements γ_1 and γ_2 are counter mappings (lifted to counter memories), via which *counter renaming* is executed. Counter renaming on the transition τ, given above, works by modifying the counter memory $\mathfrak{m}$ during a CA run. We say $\mathfrak{m}$ is *remapped* during a run; γ_1 is the *pre-update remapping* and γ_2 is the *post-update remapping*. The remapping mechanism works as follows:

1. Initially, the CA is in configuration $\langle p, \mathfrak{m} \rangle$.
2. *Before* the guards on τ are evaluated and updates are executed, $\mathfrak{m}$ is remapped according to the mapping γ_1, resulting in the memory $\mathfrak{m}_1 = \gamma_1(\mathfrak{m})$.
3. Counter guards grd_τ are evaluated on $\mathfrak{m}_1$.
4. When (and if) τ is executed, updates upd_τ are applied to $\mathfrak{m}_1$, obtaining the memory $\mathfrak{m}'_1 = \mathfrak{upd}_\tau(\mathfrak{m}_1)$.
5. *After* the update, $\mathfrak{m}'_1$ is remapped according to γ_2, obtaining the memory $\mathfrak{m}' = \gamma_2(\mathfrak{m}'_1)$.
6. The memory $\mathfrak{m}'$ becomes the new counter memory of the CA, viz. the new configuration becomes $\langle q, \mathfrak{m}' \rangle$.

Semantics of RCA. To formalize the above intuition, we again use the configuration automaton $SFA(\widehat{A})$. As with the above definition of RCA, the only thing that changes is the transition relation $\Delta_{SFA(\widehat{A})}$. It is defined as

$$\langle p, \mathfrak{m} \rangle \text{-}(\alpha)\!\!\rightarrow \langle q, \mathfrak{m}' \rangle \in \Delta_{SFA(\widehat{A})}$$

iff

$$\exists\, p \text{-}(\alpha, \gamma_1, upd_\tau, grd_\tau, \gamma_2)\!\!\rightarrow q \in \Delta\colon\ \mathfrak{grd}_\tau(\gamma_1(\mathfrak{m})) \ \wedge\ (\,\gamma_2(\mathfrak{upd}_\tau(\gamma_1(\mathfrak{m}))) = \mathfrak{m}'\,).$$

The language of a RCA $\widehat{A}$ is defined analogously to CA via $SFA(\widehat{A})$. Observe that CA are in fact a special kind of RCA, where on each transition, both γ_1 and γ_2 are the identity mapping γ_{ID}.

Note that some properties of CA are not directly applicable to RCA. For example, the definitions of counter scope and liveness would need to be modified to consider counter remapping. However, we do not need these properties for an RCA and thus we do not define them.

Now we can proceed to describe the method of merging simulation-equivalent states.

Definition 4.0.2 (Reduction of CA by simulation equivalence). Let $\sqsubseteq$ be a consistent simulation on a CA A. We can *reduce* A by $\sqsubseteq$ by merging $\sqsubseteq$-equivalent states, resulting in the *reduced RCA* $\widehat{A}$. This reduction works as follows:

1. Convert the CA A to a RCA $\widehat{A}$ by changing each transition $p\text{-}(\alpha, upd, grd)\!\!\rightarrow q$ to $p\text{-}(\alpha, \gamma_{\text{ID}}, upd, grd, \gamma_{\text{ID}})\!\!\rightarrow q$.
2. Enumerate all maximal equivalence classes of $\sqsubseteq$ (that is, classes which are not contained in other classes). For each maximal equivalence class R of $\sqsubseteq$:
   (a) Select an arbitrary state $r \in R$ as its representative.
   (b) For all the remaining states $q \in R \setminus \{r\}$ (in no particular order):

i. Let γ_q^r be the mapping from r to q and γ_r^q its inverse.

ii. Modify each incoming transition of q in $\widehat{A}$, $\tau_q \colon q' \text{-}(\alpha, \gamma_1, upd, grd, \gamma_2)\text{-}\!\!\rightarrow q$:
   A. Change the target state from q to r.
   B. Change γ_2 to $\gamma_r^q \circ \gamma_2$.

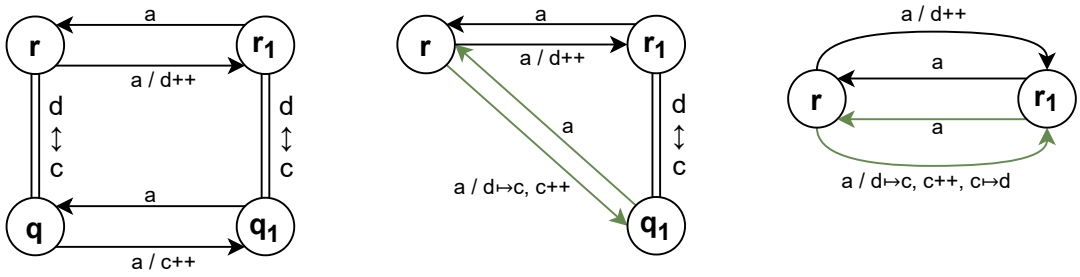
   The rename on an incoming transition τ is added *after* all existing operations and renames on τ.

iii. Modify each outgoing transition of q in $\widehat{A}$, $\tau_q \colon q \text{-}(\alpha, \gamma_1, upd, grd, \gamma_2)\text{-}\!\!\rightarrow q'$:
   A. Change the source state from q to r.
   B. Change γ_1 to $\gamma_1 \circ \gamma_q^r$.

   The rename on an outgoing transition τ is added *before* all existing operations and renames on τ.

3. Let $\widehat{A}$ be the resulting reduced RCA.

Observe that counter updates and guards do not change at all. Instead, all necessary changes are handled via the mappings γ_1 and γ_2. In particular, observe how self-loops on q are processed. First, the target changes from q to r with the rename from q to r being added to γ_2. Then, the source changes from q to r with the rename from r to q being added to γ_1. The resulting transition will start in r with some memory $\mathfrak{m}$, then remap the memory to the context of the original state q via γ_1, execute updates on $\gamma_1(\mathfrak{m})$ within this context, and then remap the updated memory back to the context of r via γ_2.



(a) Before merging q into r. (b) After merging q into r. (c) After merging q_1 into r_1.

Figure 4.1: Two successive states are merged into their representatives.

Memory context and the pre-update and post-update remappings. Having defined the semantics of memory remapping formally, we now give an illustrated intuitive overview of the principles behind it. We explain why both the pre-update remapping γ_1 and the post-update remapping γ_2 are necessary. This is best illustrated by thinking about a RCA transition as consisting of three parts: (1) the source state and its context; (2) the input predicate and counter guards and updates, which have their own context; and (3) the target state and its context. Here, context could be roughly understood as counter memory; namely it is the assignment of specific values to specific counters. This notion will become more clear in the following paragraphs.

In an original transition which has not been redirected, the contexts of source state, updates, and target state all agree. When we change the source state of a transition τ from q to r, we introduce a mismatch in τ between its source-state context and update context. We need to account for this mismatch on τ. More precisely, before the updates (or guards)

are evaluated, we need to adapt from the new source context (which is the context of r) to the context of updates (which is the context of q). This change of context consists simply of counter renaming, alias memory remapping. An obvious thought would be to simply change the updates and guards present on τ, but that is not enough. The counters still need to be renamed.[19]

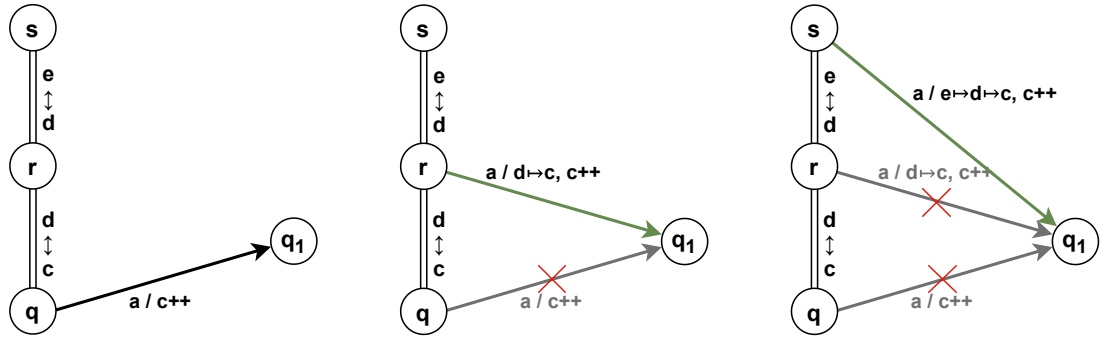
Instead, we account for the context mismatch by executing the appropriate *memory remapping*. This remapping is intended to change the context back to q, and is executed as the *first thing* on τ, before anything else (including other existing remappings on τ). After we change context from r to q, we execute the rest of pre-update remappings on τ (if there are any), then the updates, and then the original post-update remappings. We do not need to add a post-update remapping if we only change the source state of τ, since the target context of τ stays the same as before. Similarly, if we change the target state (and thus target context) of τ, we need to add a new post-update remapping. This remapping will be executed as the *last thing* on τ, after all existing pre-update remappings, updates, and existing post-update remappings. Figure 4.1 illustrates redirection of a transition between two states, both of which are merged into their representative state.

Composition of memory remappings. Observe that in point 2(b)iiB of Def. 4.0.2, the post-update remapping γ_r^q is "appended" to the existing remappings γ_2 when changing the target of a transition. On the other hand, in point 2(b)iiiB (when changing the source), the pre-update remapping γ_q^e is "prepended" to the existing remappings γ_1. This is not arbitrary: it allows for a "layered" changing of context. That means we can redirect a transition multiple times and still obtain a valid, equivalent RCA (assuming the merged states are language-equivalent). In a broader sense, it allows us to take an already-reduced RCA and reduce it again, using the same merging mechanism.

Now we illustrate more closely how composition of remappings relates to the repeated redirection of a transition. Imagine τ was originally going into a state q, from which its target state has been changed to r, and then again to s. Its target context thus has to be changed twice after executing its updates – first from q to r, then from r to s. It should be clear now that the second post-update mapping – γ_s^r – has to be executed *after* the first one – γ_r^q. Analogously in the case when changing the source state of τ, as illustrated in fig. 4.2. In fig. 4.2c, the newly added pre-update remapping $e \mapsto d$ needs to be executed *before* the original remapping $d \mapsto c$.

Simulation consistency. As we hinted in the beginning of section 3.5, we can only use a *consistent simulation* for merging states. This is to avoid modifying the language of the CA while merging states. For example, assume the following reflexive strongly symmetric simulation $\sqsubseteq$ on CA A: $\{(p, q, \gamma_1), (q, r, \gamma_2)\}$ (reflexive and symmetric elements omitted). This simulation is not transitive and thus not consistent. As r and p are not equivalent in $\sqsubseteq$, it is not clear what would happen to the language of A if we merged them. Therefore, we do not merge them, and require that a simulation must be transitive to be used for merging. As a consequence of transitivity, we also have strong symmetry. Assume a reflexive simulation $\sqsubseteq = \{(p, q, \gamma_1), (q, p, \gamma_2)\}$ (reflexive pairs omitted), where $\gamma_1 \neq (\gamma_2)^{-1}$. Here, it is not only difficult to see whether merging the states p and q is valid. It is also not clear how exactly should we remap memory on redirected transitions. We therefore do not allow asymmetric

[19] For example, assume $\gamma_q^r = \{d \mapsto c\}$ and τ is a transition to q which is not a self-loop. Then it is *necessary* to rename c to d when redirecting τ into r, since c is dead in r (otherwise it would be mapped).



(a) Before merging q into r. (b) After merging q into r. (c) After merging r into s.

Figure 4.2: A transition is redirected multiple times.

mappings between two equivalent states, as it would likely bring no benefit and only make matters more complicated.

4.1 Correctness of Merging

In this section we give a sketch of proof showing that the merging of language-equivalent states preserves the language of a CA. We use the following assumption, which is true in the CA from [55].

Assumption 4.1.1 (Dead counters have value zero). Let A be a CA, q its state, and c a counter. If c is dead in q, then for every reachable configuration $\langle q, \mathfrak{m}\rangle$ in A, $\mathfrak{m}(c) = 0$. Put simply, if a counter c is dead in state q, then its value is always zero in q.

Correctness of merging. Assume states p and q of CA A are language-equivalent under a counter mapping γ_p^q from q to p:

$$\forall\, \mathfrak{m} \in \mathfrak{M}\colon\ \mathcal{L}(\langle q, \mathfrak{m}\rangle) = \mathcal{L}(\langle p, \gamma_p^q(\mathfrak{m})\rangle). \tag{4.1}$$

In particular, this holds if p and q are are (bi)simulation-equivalent. Assume p is to be merged into q in the reduced RCA $\widehat{A}$; denote the resulting merged state q' (to avoid confusion with the original q in CA A). Since the incoming transitions of p are redirected to q', the left language of q' is (by Assumption 4.1.1):

$$\mathcal{L}_L(q') = \{\, (\mathfrak{m}, w) \mid w \in \mathcal{L}_L(\langle q, \mathfrak{m}\rangle) \cup \mathcal{L}_L(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \,\}. \tag{4.2}$$

By that same assumption, the right language of q' is

$$\mathcal{L}_R(q') = \{\, (\mathfrak{m}, w) \mid w \in \mathcal{L}_R(\langle q, \mathfrak{m}\rangle) \cup \mathcal{L}_R(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \,\}. \tag{4.3}$$

Since p and q have equal right language under any memory, this is equivalent to

$$\mathcal{L}_R(q') = \{\, (\mathfrak{m}, w) \mid w \in \mathcal{L}_R(\langle q, \mathfrak{m}\rangle) \,\} = \{\, (\mathfrak{m}, w) \mid w \in \mathcal{L}_R(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \,\}. \tag{4.4}$$

The language via q' is then

$$\mathcal{L}_V(q') = \{\, w_L \cdot w_R \mid \exists\, \mathfrak{m}\colon (\mathfrak{m}, w_L) \in \mathcal{L}_L(q') \ \wedge\ (\mathfrak{m}, w_R) \in \mathcal{L}_R(q') \,\}. \tag{4.5}$$

Clearly, we need $\mathcal{L}_V(q')$ to be the precisely the union of $\mathcal{L}_V(p)$ and $\mathcal{L}_V(q)$. Naturally $\mathcal{L}_V(q')$ is a superset of $\mathcal{L}_V(p) \cup \mathcal{L}_V(q)$, as it contains $w_L w_R$ for all w_L, w_R coming from the same state:

$$\{\, w_L \cdot w_R \mid \exists\, \mathfrak{m} \colon w_L \in \mathcal{L}_L(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \ \wedge\ w_R \in \mathcal{L}_R(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \,\}, \tag{4.6}$$

$$\{\, w_L \cdot w_R \mid \exists\, \mathfrak{m} \colon w_L \in \mathcal{L}_L(\langle q, \mathfrak{m}\rangle) \ \wedge\ w_R \in \mathcal{L}_R(\langle q, \mathfrak{m}\rangle) \,\}. \tag{4.7}$$

We now need to ensure that also pairs w_L, w_R, where the left and right word come from different states, are still in $\mathcal{L}_V(p) \cup \mathcal{L}_V(q)$:

$$\{\, w_L \cdot w_R \mid \exists\, \mathfrak{m} \colon w_L \in \mathcal{L}_L(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \ \wedge\ w_R \in \mathcal{L}_R(\langle q, \mathfrak{m}\rangle) \,\}, \tag{4.8}$$

$$\{\, w_L \cdot w_R \mid \exists\, \mathfrak{m} \colon w_L \in \mathcal{L}_L(\langle q, \mathfrak{m}\rangle) \ \wedge\ w_R \in \mathcal{L}_R(\langle p, \gamma_p^q(\mathfrak{m})\rangle) \,\}. \tag{4.9}$$

By Equation 4.1, we can replace $\mathcal{L}_R(\langle q, \mathfrak{m}\rangle)$ by $\mathcal{L}_R(\langle p, \gamma_p^q(\mathfrak{m})\rangle)$ in Eq. 4.8 and obtain Eq. 4.6. Similarly, we can obtain Eq. 4.7 from 4.9. Therefore, the language via q' is indeed the union of languages via p and q, and the languages of $\widehat{A}$ and A are equal.

In a similar fashion, we could presumably show that the merging can also be used to reduce a CA by left-language-equivalence. However, in that case Assumption 4.1.1 would need to be revised, as we no longer care about "future" of counters (viz. liveness), but their "past". For similar reasons, the definition of counter mapping would need to be revised, as its current definition considers liveness of counters. See Appendix A for a brief discussion of left simulation in CA.

Chapter 5

Proposed Algorithm for Computing Simulations in CA

In this chapter we propose an algorithm for computing simulation preorders, simulation equivalences and bisimulation equivalences on counting automata, as described in Sections 3.3 and 3.5. The algorithm consists of two sub-algorithms, Algorithm Pseudosim and Algorithm Search, each respectively presented in the following sections.

1. **Algorithm Pseudosim** takes as input a simple CA A, and computes a certain *superset of the maximal hypersimulation* $\overline{\mathcal{S}}$ of A, which we call ***pseudosimulation*** and denote it $\lceil\sqsubseteq\rceil$. This notation hints to the fact that the computed pseudosimulation is generally *greater* than the simulation which we wish to compute – in a sense it is "rounded up", as is explained later at the end of Section 5.1.

   This algorithm performs the bulk of the work of computing the final simulation preorder. In many cases, the pseudosimulation computed by it is unambiguous – then it is the simulation preorder itself. However, if the pseudosimulation is ambiguous, we need to search for simulation preorders within it, using Algorithm Search.

   The core of this algorithm is primarily inspired by the INY algorithm presented in Sec. 2.3.3. However, the modifications are nontrivial, hence the similarities between the two algorithms are minimal. This algorithm can optionally be modified to compute a strongly symmetric superset of the maximal hyperbisimulation of A, which we call ***pseudobisimulation*** and denote it $\lceil\doteqdot\rceil$.

2. **Algorithm Search** takes as input $\lceil\sqsubseteq\rceil$ computed by Algorithm Pseudosim, and searches for simulation preorders within it. It is only necessary to use this algorithm when $\lceil\sqsubseteq\rceil$ is ambiguous, meaning it can contain several simulation preorders. From the found simulation preorders, one can choose e.g. the maximal ones, or extract simulation equivalences from them. A modification of this algorithm can take as input the pseudobisimulation $\lceil\doteqdot\rceil$ and find bisimulation equivalences within it.

The modifications of Algorithms Pseudosim and Search for computing bisimulations are described in appendix C.

Used notations. In the descriptions of Algorithms Pseudosim and Search, we use the following notations:

- $\lceil\sqsubseteq\rceil$ denotes the *pseudosimulation* computed by Algorithm Pseudosim. $\lceil\sqsubseteq\rceil(q, s)$ is the set of counter mappings from s to q in $\lceil\sqsubseteq\rceil$.

- Γ denotes the *current overapproximation* of $\lceil\sqsubseteq\rceil$ in Algorithm Pseudosim. $\Gamma_q^s \equiv \Gamma(q,s)$ is the current overapproximation of $\lceil\sqsubseteq\rceil(q,s)$.
- $\dot{\Gamma}$ denotes the *initial overapproximation* of $\lceil\sqsubseteq\rceil$ in Algorithm Pseudosim. $\dot{\Gamma}_q^s \equiv \dot{\Gamma}(q,s)$ is the initial overapproximation of $\lceil\sqsubseteq\rceil(q,s)$. For any state pair (q,s), $\dot{\Gamma}_q^s$ contains only those counter mappings, which fully satisfy *local conditions* (explained later).
- $\ddot{\Gamma}$ denotes the *overapproximation of* $\dot{\Gamma}$ in Algorithm Pseudosim, also called *over-overapproximation*. $\ddot{\Gamma}_q^s \equiv \ddot{\Gamma}(q,s)$ is the overapproximation of $\ddot{\Gamma}_q^s$. Some mappings in this set may not satisfy local conditions fully, but only partially (explained later).

5.1 Algorithm Pseudosim: Computing the Pseudosimulation

This section describes computation of the *pseudosimulation* for given CA. From this pseudosimulation, we will then obtain simulation preorders using Algorithm Search, described in the next section, 5.2.

Before we present the algorithm in its full extent, let us develop an understanding of the underlying notions. We first describe informally, on a high level, the conditions which we place on the algorithm computing the pseudosimulation, and how they will be established. These conditions follow the definition of CA simulation (Def. 3.3.1). However, we will make some simplifications based on the structure of our specific CA obtained from regexes via the construction of [55].

Local and inductive conditions. For each pair of states (q,s), where q is simulated by s, there must exist a counter mapping γ. The mapping needs to fulfill conditions per Def. 3.3.1, which can be divided into two categories: *local conditions* and *inductive conditions*.[20] Local conditions can be observed by looking just at the states q and s, their acceptance conditions and outgoing transitions, but without considering simulation between their successors. Inductive conditions, on the other hand, are concerned with successors and the mapping between them.

The definition of CA simulation does not make a clear separation between local and inductive conditions, as this would be impractical. Therefore it is not easy to see, or describe, which parts of which conditions are local and which are inductive. As a result, some conditions in the definition are both local and inductive. For example, Conditions 3.10 and 3.11 are clearly local, but so are 3.12 and 3.13, albeit it does not appear so from the definition. Condition 3.14 is inductive, while the conditions in point 3.3.1(II)4 have both local and inductive character.

Let us briefly revisit the simulation in classical FA (Def. 2.3.1). There, in contrast with CA, we do not need a counter mapping between q and s, we simply have $(q,s) \in S_R$. Condition 2.1 is local, as well as 2.2 without considering simulation of the successors (that is, only concerning the symbols on the weaker and stronger transition). In its full extent, Cond. 2.2 is the inductive condition. The algorithm INY (Alg. 1) first establishes the local conditions (lines 3 and 4), then it iteratively establishes the inductive condition (loop on lines 6-13).

[20] We purposefully avoid the term *invariant*, since these conditions are not always true during the execution of the algorithm.

Initialization phase and inductive phase. Now we will attempt to adapt this approach to computing pseudosimulation in CA. Algorithm 2 shows the basic scheme for this computation. The output of this algorithm is the pseudosimulation $\lceil\sqsubseteq\rceil$. (It corresponds to the *Sim* set of the INY algorithm.)

The algorithm starts with *initialization phase*, on lines 2-9. In the loop on line 4, we establish the overapproximation $\mathring{\Gamma}$ of $\lceil\sqsubseteq\rceil$. The overapproximation contains all mappings fulfilling the local conditions. We make an exception for reflexive pairs, i.e., (q, q), as we want to compute a *consistent* simulation. Recall that per Def. 3.5.10, a consistent simulation must be *reflexive* (as defined in Def. 3.5.4). Therefore for reflexive pairs, we must only allow the identity mapping γ_{ID}.

Then follows the *inductive phase* – lines 11-14. First, the overapproximation Γ of $\lceil\sqsubseteq\rceil$ is initialized with $\mathring{\Gamma}$ computed in the initialization phase. Then, in the loop on lines 12-13, mappings which do not fulfill the inductive conditions are iteratively removed from Γ. Removal of a mapping may cause another mapping to become invalid, meaning it has to be removed as well. At the end of the computation, Γ equals $\lceil\sqsubseteq\rceil$. If the set Γ^s_q is empty, then s cannot simulate q. If it contains multiple mappings, we need to choose one of them. The problem of multiple possible mappings will be addressed later.

Algorithm 2: High-level computation of CA pseudosimulation

Input: Counting automaton A
Output: The pseudosimulation $\lceil\sqsubseteq\rceil$ on A

1 Initialization phase:
2 **for** $q \in Q$ **do**
3 | compute the sets C_q and N_q;
4 **for** $q, s \in Q \times Q$ **do**
5 | **if** $q = s$ **then**
6 | | $\mathring{\Gamma}^s_q := \{\gamma_{\text{ID}}\}$;
7 | **else**
8 | | $\mathring{\Gamma}^s_q :=$ all total injective mappings from C_s to C_q
9 | | | which fulfill local conditions;
10 Inductive phase:
11 $\Gamma := \mathring{\Gamma}$;
12 **while** there is a mapping γ^s_q which does not fulfill inductive conditions **do**
13 | remove the mapping γ^s_q from Γ^s_q;
14 {**assert** $\Gamma = \lceil\sqsubseteq\rceil$}
15 **return** Γ;

5.1.1 Initialization Phase

In our algorithm for CA simulation, we proceed somewhat differently than outlined in Alg. 2. For practical reasons, we split the initialization phase (lines 2-9 of Alg. 2) into two parts.

- **Part 1** computes a rough overapproximation of the pseudosimulation by applying weaker local conditions. We refer to this first overapproximation as *over-overapproximation*, and denote it $\ddot{\Gamma}$. The purpose of this part is to inexpensively prune the initial pseudosimulation.
- **Part 2** refines the output of the first part. Generally speaking, it removes from $\ddot{\Gamma}$ those mappings, which do not satisfy the full local conditions. The result is denoted $\dot{\Gamma}$ and corresponds to the *initial overapproximation* $\mathring{\Gamma}$ in Alg. 2. However, is not a single set like $\mathring{\Gamma}$ – for practical reasons we use several data structures, and the result $\dot{\Gamma}$ is implicit in these structures. We then proceed to the the inductive phase, where $\dot{\Gamma}$ is further refined until convergence, at which point it equals $\lceil\sqsubseteq\rceil$.

Initialization Phase, Part 1

In the first part of the initialization phase, we consider only a subset of conditions needed for computing the initial overapproximation. As a result, we compute an overapproximation of the initial overapproximation – the over-overapproximation $\ddot{\Gamma}$. We only use conditions which are inexpensive to evaluate, and thus allow us to compute $\ddot{\Gamma}$ quickly.

Initial pruning via classical simulation $\preccurlyeq$. In a large automaton, computation of $\ddot{\Gamma}^s_q$ for each pair $(q, s) \in Q \times Q$ may be too expensive. Therefore, we want to first prune the set of possible simulating state pairs to make it significantly smaller than $Q \times Q$. Once we do that, we compute $\ddot{\Gamma}^s_q$ for each pair (q, s) in the set. We realize this pruning via the so-called *structural SFA* of the input CA A.

Definition 5.1.1 (Structural SFA). Given a CA $A = (\mathbb{I}, C, Q, q_0, F, \mathit{fin}, \Delta)$, its *structural SFA* is the SFA $A' = (\mathbb{I}, Q, q_0, F, \Delta')$, where $\exists\, p \text{-}\langle \alpha, \mathit{upd}, \mathit{grd}\rangle\!\!\rightarrow q \in \Delta \iff p \text{-}\langle \alpha\rangle\!\!\rightarrow q \in \Delta'$.

Intuitively, the structural SFA is obtained from the CA by removing counters and their guards and updates. By doing this, all conditionally final states become unconditionally final, and the executability of each transition depends solely on its input predicate. This can be seen as modifying A so that each acceptance condition as well as each guard on each transition is $\top$.

It should be clear that $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. More importantly, $p \sqsubseteq_\gamma q$ in A implies $p \preccurlyeq q$ in A' (where $\preccurlyeq$ is the classical simulation preorder on A'). Therefore if $p \not\preccurlyeq q$ in A', they cannot be in simulation in A, so we do not need to compute their $\ddot{\Gamma}^s_q$. To obtain $\preccurlyeq$, we can use any algorithm for FA simulation, e.g. INY or some of the more sophisticated algorithms [47; 17; 15].[21] Let us now focus on those (q, s) for which we do need to compute $\ddot{\Gamma}^s_q$.

[21] Recall that the input CA A is simple, therefore mintermized, and therefore A' is also mintermized. Since a mintermized SFA can be treated as a FA (in the context of simulation), we can use classical FA simulation algorithms on A'.

Algorithm 3: Computation of the over-overapproximation $\ddot{\Gamma}$

Input: States (q, s) s.t. $q \preccurlyeq s$ in A'; sets C_q, C_s; CA A
Output: $\ddot{\Gamma}^s_q$, an overapproximation of the initial mapping set $\dot{\Gamma}^s_q$

$\ddot{\Gamma}^s_q := \{\}$;
for every total injective mapping γ^s_q from C_s to C_q **do**
  $\mathit{mappingIsValid} :=$ true;
  for $(d \mapsto c) \in \gamma^s_q$ **do**
    $\mathit{subsumptionHolds} := (c \subseteq d)$;
    $\mathit{uncondAcceptImplied} := ((\mathit{fin}_q(c) = \textbf{true}) \implies (\mathit{fin}_s(d) = \textbf{true}))$;
    if not ($\mathit{subsumptionHolds}$ **and** $\mathit{uncondAcceptImplied}$) **then**
      $\mathit{mappingIsValid} :=$ false;
      break;
  if $\mathit{mappingIsValid}$ **then**
    $\ddot{\Gamma}^s_q := \ddot{\Gamma}^s_q \cup \{\gamma^s_q\}$;
return $\ddot{\Gamma}^s_q$;

Subsumption of counters. As we have noted, our algorithm is tailored for the CA obtained from regexes described in [55]. We thus make certain *assumptions* which are true for these specific CA, but may not be true in the general case. One such assumption is stated below. We use it to make the computation of counter mappings more efficient.

Assumption 5.1.1 (**Stronger counter subsumes weaker**)**.** We assume that in the input CA A, every counter d is always used in a loop. It is always incremented in this loop, and the loop always contains an exit of d, either on a transition or in the acceptance condition of some state. That means every live counter will eventually be both incremented and exited. Therefore, if $q \sqsubseteq_\gamma s$ holds in A, the mapping γ must map every stronger counter d to such a weaker counter c, that both $\textsc{canIncr}(c)$ implies $\textsc{canIncr}(d)$, and $\textsc{canExit}(c)$ implies $\textsc{canExit}(d)$. Equivalently, c is subsumed by d (written $c \subseteq d$); that is, the bounds of the weaker counter are contained within the bounds of the stronger counter.

Weakening the local conditions. Obviously, Assumption 5.1.1 can reduce the number of mappings between two states in $\lceil\sqsubseteq\rceil$. However, it also opens the door to another optimization. The idea is to *weaken the local conditions* to approximate $\lceil\sqsubseteq\rceil$ roughly and inexpensively. Namely, we ignore outgoing transitions and only consider subsumption of counters and acceptance conditions when computing the initial counter mappings. This way, we obtain an overapproximation of $\dot{\Gamma}$, denoted $\ddot{\Gamma}$, referred to as the *over-overapproximation (of $\lceil\sqsubseteq\rceil$)*. We use it in an additional pruning by $\preccurlyeq$, discussed further below. The simplified computation of $\ddot{\Gamma}$ is illustrated by Algorithm 3.

This algorithm checks for each possible mapping γ^s_q that (1) each stronger counter d subsumes its mapped weaker counter c, and (2) if q exits *unconditionally* w.r.t. some c, then s exits unconditionally w.r.t. the mapped d. Notice we do not explicitly check Condition 3.11. It is not necessary, as it is implied by the conjunction of (1) and (2).

Computing initial mappings efficiently. We now propose an optimization of Algorithm 3, as its approach is rather naïve. It examines *all* possible total injections from C_s to C_q, but can end up discarding most of them because they do not fulfill the necessary conditions. For automata with many counters, the number of examined mappings (which

is $\Omega(|C_s|!)$) may be significant. It would be more efficient to only generate those mappings which will certainly be valid. We can accomplish this as follows.

First, we construct a bipartite graph $G^s_q = (C_s \cup C_q,\ E)$, where $(c, d) \in E$ iff c and d fulfill the conditions *subsumptionHolds* and *uncondAcceptImplied* of Alg. 3. Then, we search for all matchings in this graph which cover the set C_s. Each found matching corresponds to a mapping γ^s_q from C_s to C_q, and by construction it fulfills the conditions of Alg. 3. Hence the set of all such matchings in G^s_q is the set $\ddot{\Gamma}^s_q$. If the set C_s is empty, then a single *empty matching* is produced, instead of no matching. This is important, as the existence of a mapping γ^s_q (albeit empty) is necessary for s to simulate q.

Additional pruning of $\ddot{\Gamma}$ via $\preccurlyeq$. We have computed the initial over-overapproximation $\ddot{\Gamma}$ for each $q \preccurlyeq s$. However, for some (q, s) there may be no mappings, viz. $\ddot{\Gamma}^s_q = \emptyset$. Based on this, we can prune $\ddot{\Gamma}$ further, by re-using the simulation $\preccurlyeq$ on $SFA(A)$. Let $\not\Gamma$ denote the set of all pairs (q, s) with no mapping, viz. $\ddot{\Gamma}^s_q = \emptyset$. If a pair (q, s) has no mapping, then it cannot be in simulation. Due to this, we need to remove from $\ddot{\Gamma}$ all predecessor pairs (p, r) which depend on $q \sqsubseteq s$.[22] We can remove some of these pairs via the computed $\preccurlyeq$ as follows.

First, we assert that $q \not\preccurlyeq s$ for all $(q, s) \in \not\Gamma$. Then we run the inductive computation of $\preccurlyeq$ again until convergence. The next paragraph describes this in more detail. The pairs newly removed from $\preccurlyeq$ during the re-computation will then be removed also from $\ddot{\Gamma}$. This way, we remove from $\ddot{\Gamma}$ all pairs (p, r) without a simulating successor pair w.r.t. *input predicates*. Per Cond. 3.12, such a successor pair is necessary for $p \sqsubseteq r$ to hold. Therefore, all of the removed pairs should indeed be removed from $\ddot{\Gamma}$ (i.e., we do not lose any pairs which could actually be in simulation).

Let us now describe the technique of re-computing of $\preccurlyeq$. The exact way of performing this operation efficiently depends on the used algorithm. We use INY (Alg. 1) for illustration. The key principle is to resume computation with the existing precomputed data structures. Firstly, we need to save the relevant internal state upon return from the first invocation. In the case of INY we need to preserve the current values of *Sim* and N_a (for each $a \in \Sigma$). Secondly, we must be able to supply *NotSimQueue* externally, e.g. via a parameter.

After computing $\not\Gamma$, we want to "assert" that $q \not\preccurlyeq s$ for each (q, s) in $\not\Gamma$. We do this by invoking the INY algorithm again, using $\not\Gamma$ as *NotSimQueue*. However, this time we continue the execution from the loop on line 6. Let $\overline{\preccurlyeq}$ denote the result of this second invocation. This is the maximal simulation on A' which does not contain any of the pairs in $\not\Gamma$. After we obtain $\overline{\preccurlyeq}$, we remove from $\ddot{\Gamma}$ all pairs in $Q \times Q \setminus \overline{\preccurlyeq}$. (As an optimization, we could modify the algorithm to return the newly removed pairs along with *Sim*. This would simplify updating $\ddot{\Gamma}$, as we no longer have to iterate over all pairs in $Q \times Q$.)

End of part 1. The first part of the initialization phase is now over. Its output is the over-overapproximation $\ddot{\Gamma}$. In the second part, we prune it further to obtain $\dot{\Gamma}$ (albeit implicitly stored in several data structures).

[22] By removing (p, r) from $\ddot{\Gamma}$ we mean removing all mappings from the set $\ddot{\Gamma}^r_p$.

Initialization Phase, Part 2

The second part of the initialization phase further refines the approximation $\ddot{\Gamma}$ obtained in the first part. Unlike in the first part, we now consider outgoing transitions in the original CA A. By doing this, we are able to ensure local conditions in their full, strongest form. The result is $\dot{\Gamma}$ – the initial approximation to be used in the inductive phase. It roughly agrees with the initial approximation $\mathring{\Gamma}$ computed by Algorithm 2 (lines 2-9). However, as we applied multiple prunings in part 1, $\dot{\Gamma}$ is a subset of $\mathring{\Gamma}$.

Simulation of transitions. Now that we have sufficiently pruned the sets $\ddot{\Gamma}^s_q$, we are ready to consider simulation of transitions. That allows us to ensure the full local conditions. However, it is impractical to consider exclusively local conditions. We also want to consider successor mappings, as the very definition of simulation requires it. We will thus do the following for each γ^s_q in $\ddot{\Gamma}$: find for each weaker transition τ_q all the stronger transitions τ_s, which simulate τ_q per definition of simulation (Def. 3.3.1(II)). More precisely, for each τ_q, we find all *simulating transitions* τ_s and *successor mappings* $\gamma^{s'}_{q'}$ for τ_s. This follows from the definition of CA simulation, which states that each weaker transition τ_q needs a stronger transition τ_s and a successor mapping $\gamma^{s'}_{q'}$.

Transition matches. Following the definition of CA simulation, we must ensure that if s simulates q under a mapping γ^s_q, then each weaker transition τ_q has a stronger transition τ_s and a successor mapping $\gamma^{s'}_{q'}$. Thus we associate with each pair (γ^s_q, τ_q) a set of pairs $(\tau_s, \gamma^{s'}_{q'})$. We call the pair $(\tau_s, \gamma^{s'}_{q'})$ a *transition match* (or simply *match*) of τ_q under the (current) mapping γ^s_q. It should be noted that the states s and q in upper and lower index of γ^s_q are important, as the same mapping can exist between two different pairs of states. We thus need to distinguish e.g. (γ^s_q, τ_q) from (γ^r_q, τ_q), even if the mappings γ^s_q and γ^r_q differ only by the stronger state and are otherwise equal.

***Tmatches* – data structure for transition matches.** If under a mapping γ^s_q there is a τ_q with no matches, then it has no simulating stronger transition. Therefore, s cannot simulate q under the mapping γ^s_q, hence γ^s_q is invalid and has to be removed from $\ddot{\Gamma}$.

We use the *Tmatches* structure to store and manipulate transition matches, in order to detect that a mapping does not satisfy conditions and needs to be removed. This structure is an associative array, where each pair (γ^s_q, τ_q) is mapped to a set of its transition matches $(\tau_s, \gamma^{s'}_{q'})$. Symbolically,

$$\mathit{Tmatches}\colon (\gamma^s_q, \tau_q) \mapsto \{(\tau_s, \gamma^{s'}_{q'})\}.$$

We denote by $\mathit{Tmatches}[\gamma^s_q, \tau_q]$ the set of matches associated with (γ^s_q, τ_q). This structure is dynamically updated during the execution of the algorithm, as we remove counter mappings which violate inductive conditions. The next section (Sec. 5.1.2) discusses this in more detail.

Computing initial transition matches. Algorithm 4 illustrates calculation of transition matches for a single τ_q under a counter mapping γ^s_q. It is not listed in its entirety as it straightforwardly corresponds with the (very lengthy) definition of CA simulation (Def. 3.3.1(II)), to which we refer on line 4.

Algorithm 5 shows initialization of the *Tmatches* structure. It uses Alg. 4 to compute transition matches for each weaker transition. If a weaker transition has no match under some mapping, then this mapping is removed. This is not done directly, however, as

Algorithm 4: `Find-Transitions-Matches`
(Finding transition matches for a weaker transition)

Input: Mapping γ_q^s from s to q; transition τ_q from q to q';
over-overapproximation $\ddot{\Gamma}$; CA A
Output: The set of transition matches $(\tau_s, \gamma_{q'}^{s'})$ for τ_q under γ_q^s

```
1 transitionMatches := {};
2 for each transition τ_s from s to s' do
3 |   for γ_{q'}^{s'} ∈ Γ̈_{q'}^{s'} do
4 |   |   if τ_s can simulate τ_q under mapping γ_q^s and successor mapping γ_{q'}^{s'}
  |   |    according to Definition 3.3.1(II) then
5 |   |   |   transitionMatches := transitionMatches ∪ {(τ_s, γ_{q'}^{s'})};
6 return transitionMatches;
```

removing a mapping now induces some maintenance operations. It therefore has to be done via the queue *CmapRemoveQueue*. This is a queue of mappings to be removed, principally the same as *NotSimQueue* in the INY algorithm. This queue cannot contain duplicate elements, and as with *Tmatches*, the states of a mapping are a part of the mapping itself (e.g. two mappings which differ only by states are *not* considered equal). Note that although it is possible to apply pruning by $\preccurlyeq$ again, the improvement in efficiency would likely be very limited; hence it is not done. The purpose of the data structure *SuccCmapIndex* is explained in the following paragraph.

Removal of counter mappings via *SuccCmapIndex*. We will later present the inductive phase of the algorithm, where counter mappings are iteratively *removed*, until all mappings fulfill the inductive conditions. As hinted above, removal of a mapping does not simply consist of removing it from Γ (or $\ddot{\Gamma}$). We also need to maintain the *Tmatches* structure. Intuitively, if we find out that a mapping $\gamma_{q'}^{s'}$ is no longer valid (i.e., about to be removed), we need to remove matches which use it as their successor mapping. In particular, for each mapping $\gamma_{q'}^{s'}$ removed from Γ, we need to remove $(\tau_s, \gamma_{q'}^{s'})$ from $\mathit{Tmatches}[\gamma_q^s, \tau_q]$ for every γ_q^s, τ_q and τ_s.

However, it would be inefficient if we were to search through the entire *Tmatches* structure, looking for all those matches where $\gamma_{q'}^{s'}$ is the successor mapping. We therefore employ an *index* on $\gamma_{q'}^{s'}$, *SuccCmapIndex*, which will speed up this search. It is an associative array which assigns to each successor counter mapping $\gamma_{q'}^{s'}$ the set of triples $(\gamma_q^s, \tau_q, \tau_s)$ which depend on it, viz. $(\tau_s, \gamma_{q'}^{s'}) \in \mathit{Tmatches}[\gamma_q^s, \tau_q]$. It is initialized in Algorithm 5 on lines 9-10.

End of part 2. The initialization phase is now concluded. Its output is the *Tmatches* structure, as well as the pre-populated queue *CmapRemoveQueue* and index *SuccCmapIndex*. The initial overapproximation $\dot{\Gamma}$ is implicitly stored in $\ddot{\Gamma}$ and *CmapRemoveQueue*: $\dot{\Gamma} = \ddot{\Gamma} \setminus \mathit{CmapRemoveQueue}$. Now we can proceed with the inductive phase, in which we prune $\dot{\Gamma}$ to obtain the pseudosimulation $\lceil \sqsubseteq \rceil$.

Algorithm 5: Finding transition matches under all mappings

Input: Over-overapproximation $\ddot{\Gamma}$; CA A
Output: Initialized *Tmatches*, *SuccCmapIndex*, and *CmapRemoveQueue*; initial overapproximation $\dot{\Gamma}$ (implicit in $\ddot{\Gamma}$ and *CmapRemoveQueue*)

```
1  Tmatches := empty Tmatches data structure;
2  SuccCmapIndex := empty SuccCmapIndex data structure;
3  CmapRemoveQueue := empty queue of counter mappings not allowing duplicates;
4  for q, s ∈ Q × Q do
5  |  for γ_q^s ∈ Γ̈_q^s do
6  |  |  for each transition τ_q from q do
7  |  |  |  Tmatches[γ_q^s, τ_q] := Find-Transitions-Matches(γ_q^s, τ_q);
8  |  |  |  // add the matches to the successor mapping index
9  |  |  |  for (τ_s, γ_q'^s') ∈ Tmatches[γ_q^s, τ_q] do
10 |  |  |  |  SuccCmapIndex[γ_q'^s'] := SuccCmapIndex[γ_q'^s'] ∪ {(γ_q^s, τ_q, τ_s)};
11 |  |  |  if Tmatches[γ_q^s, τ_q] = ∅ then
12 |  |  |  |  CmapRemoveQueue := CmapRemoveQueue ∪ {γ_q^s};
13 {assert Γ̈ \ CmapRemoveQueue = Γ̇}
14 return Tmatches, SuccCmapIndex, CmapRemoveQueue;
```

5.1.2 Inductive Phase

Now we have initialized the internal state of Algorithm Pseudosim and we are ready to commence the inductive phase. First, we let the overapproximation Γ equal the $\ddot{\Gamma}$ computed in the initialization phase. Then we iteratively refine Γ until convergence, at which point it equals the resulting pseudosimulation $\lceil\sqsubseteq\rceil$. In a nutshell, this phase repeats two steps: removing a counter mapping, and removing all transition matches which depend on this mapping.

The main loop. Algorithm 6 shows pseudocode of the inductive phase. The queue *CmapRemoveQueue* contains mappings which do not fulfill the inductive conditions (so-called *invalid mappings*). These mappings have to be removed from Γ. This is done in the *main loop* starting on line 3. It is executed until the queue *CmapRemoveQueue* is empty, at which point Γ equals the desired pseudosimulation $\lceil\sqsubseteq\rceil$.

Invariants. Let us now present invariants holds at the beginning and end of each iteration of the main loop. They help us grasp mutual relations between the individual data structures. We will refer to these invariants later on. In the following, let

1. $\tau_q = q\text{-}(\alpha, upd_{\tau_q}, grd_{\tau_q})\!\!\rightarrow q'$,
2. $\tau_s = s\text{-}(\beta, upd_{\tau_s}, grd_{\tau_s})\!\!\rightarrow s'$,
3. γ_q^s be a mapping from s to q,
4. $\gamma_{q'}^{s'}$ be a mapping from s' to q'.

Invariant 1 (*Tmatches*). $(\tau_s, \gamma_{q'}^{s'}) \in Tmatches[\gamma_q^s, \tau_q]$ iff τ_s can simulate τ_s under mapping $\gamma_{q'}^{s'}$ and successor mapping $\gamma_{q'}^{s'}$. More precisely, in the current approximation Γ of the pseudosimulation, τ_s is a *simulating transition* of τ_q under current mapping γ_q^s and successor

Algorithm 6: Inductive computation

Input: Over-overapproximation $\ddot{\Gamma}$; initialized *Tmatches*, *SuccCmapIndex*, and *CmapRemoveQueue*; CA A

Output: Pseudosimulation $\lceil\sqsubseteq\rceil$

```
1  {assert Γ̈ \ CmapRemoveQueue = Γ̇}
2  Γ := Γ̈;
3  while CmapRemoveQueue ≠ ∅ do
4      {assert Invariants 1, 2 and 3 hold}
5      remove some γ_q^s from CmapRemoveQueue;
6      Γ_q^s := Γ_q^s \ {γ_q^s};
7      // remove matches where γ_q^s is the current mapping
8      for each transition τ_q from q do
9          for (τ_s, γ_{q'}^{s'}) ∈ Tmatches[γ_q^s, τ_q] do
10             Remove-Tmatch(γ_q^s, τ_q, τ_s, γ_{q'}^{s'});
11     // remove matches where γ_q^s is the successor mapping
12     for (γ_{q'}^{s'}, τ_q, τ_s) ∈ SuccCmapIndex[γ_q^s] do
13         {assert Invariant 2 holds}
14         Remove-Tmatch(γ_{q'}^{s'}, τ_q, τ_s, γ_q^s);
15         if Tmatches[γ_{q'}^{s'}, τ_q] = ∅ then
16             // last match removed for weaker transition τ_q – remove γ_{q'}^{s'}
17             CmapRemoveQueue := CmapRemoveQueue ∪ {γ_{q'}^{s'}};
18 {assert Γ = ⌈⊑⌉}
19 return Γ;
20 procedure Remove-Tmatch(γ_q^s, τ_q, τ_s, γ_{q'}^{s'})
21     {assert Invariant 2 holds}
22     Tmatches[γ_q^s, τ_q] := Tmatches[γ_q^s, τ_q] \ {(τ_s, γ_{q'}^{s'})};
23     SuccCmapIndex[γ_{q'}^{s'}] := SuccCmapIndex[γ_{q'}^{s'}] \ {(γ_q^s, τ_q, τ_s)};
24     {assert Invariant 2 holds}
```

mapping $\gamma_{q'}^{s'}$ per Def. 3.3.1(II). In such case, we say $(\tau_s, \gamma_{q'}^{s'})$ is a *valid* transition match of (γ_q^s, τ_q). As a consequence, if $(\tau_s, \gamma_{q'}^{s'}) \in \mathit{Tmatches}[\gamma_q^s, \tau_q]$, then $\gamma_q^s \in \Gamma_q^s$ and $\gamma_{q'}^{s'} \in \Gamma_{q'}^{s'}$.

Invariant 2 (*SuccCmapIndex*). *SuccCmapIndex* contains precisely the currently valid transition matches: $(\gamma_q^s, \tau_q, \tau_s) \in \mathit{SuccCmapIndex}[\gamma_{q'}^{s'}]$ iff $(\tau_s, \gamma_{q'}^{s'}) \in \mathit{Tmatches}[\gamma_q^s, \tau_q]$.

Moreover, Invariant 2 holds before and after each execution of the `Remove-Tmatch` procedure in Algorithm 6. (The purpose of this procedure is to ensure this invariant.)

Invariant 3 (*CmapRemoveQueue*). For all γ_q^s in *CmapRemoveQueue*, s cannot simulate s under γ_q^s. In particular, there exists an outgoing transition τ_q from q, such that $\mathit{Tmatches}[\gamma_q^s, \tau_q] = \emptyset$.

Moreover, Invariant 3 holds at all times.

Loop body. In the body of the main loop, we remove a mapping γ_q^s from *CmapRemoveQueue*. This mapping has been found to violate inductive conditions, and is thus removed from Γ (line 6). This may invalidate Invariant 1. We then re-establish this invariant by removing transition matches which depend on γ_q^s. By doing so, we may invalidate other mappings, which are then put into *CmapRemoveQueue*.

Let us give a more detailed description. Generally, the loop body consists of two aspects: (1) *maintenance*, which ensures that all transition matches are valid (i.e., ensures Invariants 1 and 2); and (2) *propagation*, which ensures that newly-invalidated mappings are put into the *CmapRemoveQueue* and eventually removed. We now elucidate these aspects in more detail:

1. *Maintenance.*

   For the removed mapping γ_q^s, we remove from *Tmatches* all matches which depend on it one way or another. That means we remove both (1) matches where γ_q^s is the current mapping – lines 8-10; and (2) matches where γ_q^s is the successor mapping – lines 12-17. This re-establishes Invariant 1, but may invalidate Invariant 2. We therefore also need remove these matches from *SuccCmapIndex*, as they no longer exist. That is why we use the procedure `Remove-Tmatch` for removing matches. This procedure maintains Invariant 2 – it is both its precondition and postcondition.[23]

   This maintenance is not merely an optimization. It is essential for the correct working of the algorithm. If we omitted it, the algorithm may never terminate.

2. *Propagation.*

   Assume that during maintenance we removed the *last match* of a weaker transition τ_q under a mapping $\gamma_{q'}^{s'}$. This means τ_q no longer has a simulating stronger transition. Therefore, $\gamma_{q'}^{s'}$ cannot be a valid mapping, so we add it to *CmapRemoveQueue* to be removed (lines 15-17). By doing so, we ensure Invariant 3 and we do not affect Invariants 1 and 2.

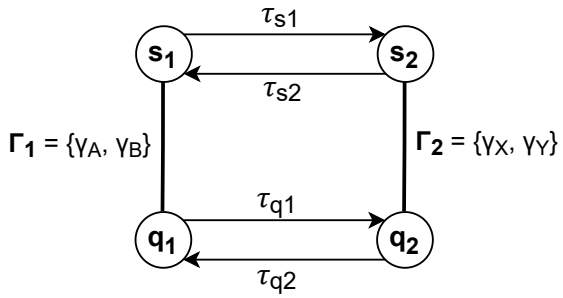
At the end of the inductive phase, we have removed all mappings which were found to violate inductive conditions. Thus we obtain the resulting pseudosimulation $\lceil\sqsubseteq\rceil$. Let us now address the problem of termination of the main loop.

Termination. The loop terminates once *CmapRemoveQueue* is empty. This will necessarily happen, as there are a finite number of mappings in Γ and no mapping will enter *CmapRemoveQueue* twice. For the sake of contradiction, assume a mapping does enter the queue twice. It is clear from *CmapRemoveQueue* being a set, that it cannot contain the same mapping twice at the same time. Thus a mapping needs to be added, removed, and then added again. To see why this cannot happen, consider the following.

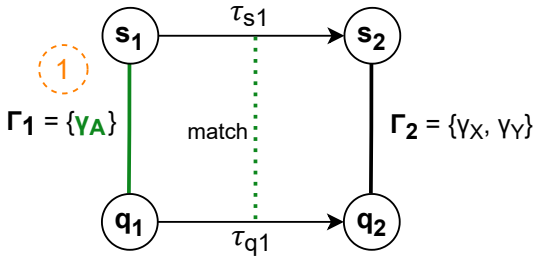
A new mapping $\gamma_{q'}^{s'}$ is added into *CmapRemoveQueue* only if we *removed the last match* of some τ_q under $\gamma_{q'}^{s'}$ (lines 15-17). Namely, if this match was already removed before (e.g. in the previous loop iteration), we do not add $\gamma_{q'}^{s'}$ into *CmapRemoveQueue*. This is owing to the fact that Invariant 2 holds in the loop on line 15. This means the loop is executed only for matches which still exist (viz. have not been removed yet).

Assume now that $\gamma_{q'}^{s'}$ is to be put into the queue for a second time. Since it has already been removed from the queue before, it has no matches (we removed them during

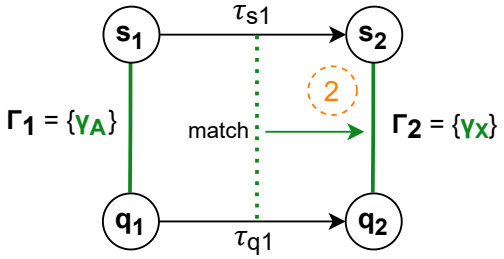
[23] Meaning it holds before and after execution of this procedure.



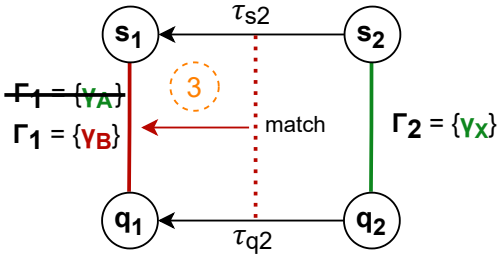
(a) Initial pseudosimulation computed by Algorithm Pseudosim.



(b) Step 1: disambiguating Γ_1 by selecting γ_A as the mapping from s_1 to q_1.



(c) Step 2: disambiguation of Γ_1 causes Γ_2 to contain only γ_X.



(d) Step 3: γ_X in Γ_2 requires Γ_1 to contain γ_B, but we already selected γ_A.

Figure 5.1: Pseudosimulation which is not a hypersimulation. It contains an invalid mapping γ_A from s_1 to q_1, which cannot exist in a hypersimulation.

maintenance). Thus the loop on line 15 will not be executed for any match under $\gamma^{s'}_{q'}$, and therefore $\gamma^{s'}_{q'}$ will *not* be added into *CmapRemoveQueue* again – contradiction.

We have now shown that each mapping will enter the queue at most once, and as a result, the algorithm will terminate after a finite number of iterations of the main loop.

Output of Algorithm Pseudosim. As we have stated before, Algorithm Pseudosim may not compute the maximal hypersimulation $\overline{\mathcal{S}}$ of the input CA. In general, it computes an overapproximation of $\overline{\mathcal{S}}$ – the pseudosimulation $\lceil\sqsubseteq\rceil$. Let us explain why this pseudosimulation may not be a hypersimulation.

We compute the pseudosimulation $\lceil\sqsubseteq\rceil$ by overapproximating the desired hypersimulation $\overline{\mathcal{S}}$, and then iteratively removing mappings found to be invalid. There may be cases, however, when we cannot discover that a mapping is invalid. As a result, the pseudosimulation may contain invalid mappings, which are not present in $\overline{\mathcal{S}}$. This is illustrated in the following example.

Example 5.1.1. Fig. 5.1a shows a pseudosimulation $\lceil\sqsubseteq\rceil$ computed by Algorithm Pseudosim. There, set Γ_1 of mappings from state s_1 to q_1 contains multiple mappings, and so does set Γ_2 from state s_2 to q_2. This means $\lceil\sqsubseteq\rceil$ is ambiguous. In order to obtain a consistent simulation, we must *disambiguate* it, viz. select a single mapping in Γ_1 and Γ_2. In step 1 (Fig. 5.1b) we disambiguate Γ_1 by selecting γ_A. Under this mapping, τ_{s1} simulates τ_{q1} only with successor mapping γ_X in Γ_2. In step 2 (Fig. 5.1c), we thus select γ_X in Γ_2. But under γ_X, τ_{s2} simulates τ_{q2} only when the mapping in Γ_1 is γ_B. Since we already selected $\Gamma_1 = \{\gamma_A\}$, we cannot have $\Gamma_1 = \{\gamma_B\}$. Therefore, our premise that s_1 can simulate q_1

under γ_A is wrong. γ_A is thus an *invalid* mapping which cannot exist in a hypersimulation; hence $\gamma_A \notin \overline{\mathcal{S}}$, and hence $\overline{\mathcal{S}} \subset \lceil\sqsubseteq\rceil$. We note, however, that this example is contrived. It is not known to us whether such situation could happen in practice.

End of inductive phase. This concludes the inductive phase, and the entire Algorithm Pseudosim. The outputs are the following:

1. the pseudosimulation $\lceil\sqsubseteq\rceil$, which contains all mappings fulfilling the inductive conditions (and possibly some which do not),
2. the *Tmatches* and *SuccCmapIndex* data structures, which will be used in further processing.

The computed pseudosimulation $\lceil\sqsubseteq\rceil$ is reflexive, as each state simulates itself under the identity mapping γ_{ID}. However, $\lceil\sqsubseteq\rceil$ may be ambiguous, meaning it allows multiple mappings between a single state pair (q, s). Further, it may not be transitive (recall Lemma 3.5.3). For merging, we need a hypersimulation which unambiguous and transitive, hence a *consistent simulation* (Def. 3.5.10). We can thus check whether $\lceil\sqsubseteq\rceil$ is unambiguous and transitive. If yes, then $\lceil\sqsubseteq\rceil$ is the unique simulation preorder of the input CA, and we can directly use it for reduction. If not, we need to *search* for consistent simulations within $\lceil\sqsubseteq\rceil$. This is done by the Algorithm Search, presented in the following section.

5.2 Algorithm Search: Searching for Consistent Simulations in the Pseudosimulation

Using Algorithm Pseudosim, we computed the pseudosimulation $\lceil\sqsubseteq\rceil$ of the input CA A. Theorem 3.5.6 tells us that A may have several simulation preorders. These are all contained within $\lceil\sqsubseteq\rceil$, and we need to *search* for them. This is the purpose of Algorithm Search. It finds within $\lceil\sqsubseteq\rceil$ simulations which are *consistent*, viz. reflexive, transitive, and unambiguous. The simulation preorders of A are then the *maximal ones* among these consistent simulations (per Def. 3.5.11). Since each state can simulate itself under the identity mapping, $\lceil\sqsubseteq\rceil$ is always reflexive. If it is also unambiguous and transitive, then it is the unique simulation preorder of A. No further processing is needed; we can simply use $\lceil\sqsubseteq\rceil$ for reduction of A. If it is not, we need to extract simulations from $\lceil\sqsubseteq\rceil$ while ensuring their *unambiguity* and *transitivity*. Let us consider the two cases separately: in the first case, we only need to face the problem of ambiguity of $\lceil\sqsubseteq\rceil$; in the second case, non-transitivity.

Case 1: Pseudosimulation is transitive but *ambiguous*.

Given an ambiguous pseudosimulation which is transitive, we must *disambiguate* it by selecting a single mapping between each state pair (in one direction). For example, if $\lceil\sqsubseteq\rceil(q, s) = \{\gamma_1, \gamma_2\}$ and $\lceil\sqsubseteq\rceil(s, q) = \{\gamma_3, \gamma_4\}$, a possible disambiguation is $\lceil\sqsubseteq\rceil(q, s) = \{\gamma_1\}$ and $\lceil\sqsubseteq\rceil(s, q) = \{\gamma_3\}$.

We can accomplish this by removing mappings which cause ambiguity. At the same time, we need to ensure that the result is still a valid simulation. In particular, after removing a mapping, we need to propagate its removal by removing other mappings which depend on it, as is done in Algorithm 6 after line 6. Thus a possible strategy would be to select an arbitrary pair (q, s) which has several mappings (i.e., $|\lceil\sqsubseteq\rceil(q, s)| > 1$) and disambiguate it by selecting an arbitrary single mapping in $\lceil\sqsubseteq\rceil(q, s)$ and removing all others. Unfortunately, this approach will not work in such

cases as described in Example 5.1.1. In the disambiguation described therein, we made a mistake by selecting a mapping between two states which turned out to be invalid. We then need to return (*backtrack*) to the point where we made the invalid choice, and make a different choice. Such a backtracking search will find all maximal unambiguous simulations in $\lceil\sqsubseteq\rceil$. However, it is not guaranteed that they are transitive. If a found simulation is *not* transitive, we need to search for maximal transitive fragments[24] within it. This means, even if we know $\lceil\sqsubseteq\rceil$ is transitive, we still need to solve the problem of transitivity. This problem is addressed in the following paragraphs. Albeit it is described for a pseudosimulation, it applies equally to a simulation.

Case 2: Pseudosimulation is unambiguous but *not transitive*.

Given a pseudosimulation which is unambiguous but not transitive, we must search for *maximal transitive fragments (MTFs)* within it. Lemma 3.5.7 tells us that these are precisely the simulation preorders of the input CA. This problem is NP-hard, as we will show in the following.

Assume a pseudosimulation $\lceil\sqsubseteq\rceil$ which contains at most one mapping between any two states – the identity mapping γ_{ID}. This pseudosimulation can then be reduced to a binary relation $R \subseteq Q \times Q$, by considering only states and ignoring the mapping γ_{ID} which is always the same. Finding MTFs of $\lceil\sqsubseteq\rceil$ is at least as hard as finding *maximal transitive subrelations* of R.[25] This problem trivially reduces to finding *maximal transitive subgraphs* of a directed graph: Q is the set of vertices and R is the set of edges. This problem is known to be NP-hard [59]. We therefore do not attempt to solve this problem efficiently for the following reasons. Firstly, we do not expect to face this problem often. In other words, we expect the pseudosimulations found by Algorithm Pseudosim (as well as the unambiguous simulations within it) to be mostly transitive. Secondly, this problem has been already studied (e.g. [11; 10]). Thirdly, even if we attempted an efficient solution, it would be quite complex and still of exponential complexity the worst case.

Example 5.2.1 (**Sketch of Algorithm Search**)**.** Algorithm 7 presents the general idea of a *backtracking search* for consistent simulations within a pseudosimulation. We will not be using Algorithm 7 as the actual Algorithm Search, but the principles are the same. We present it merely as a sketch to illustrate the approach in a simplified way; it is therefore unoptimized and described somewhat unrigorously (specifically on line 10 we refer to the *main loop* of Alg. 6).

Essentially, the algorithm proceeds as follows. It tries all possible ways of disambiguating the pseudosimulation. The result is several unambiguous simulations. In each of these, it finds maximal transitive fragments.[24] Lastly, it finds the maximal across *all* found transitive fragments. These are the simulation preorders – the output of the algorithm.

Let us examine the algorithm in more detail. Its input is a pseudosimulation $\lceil\sqsubseteq\rceil$, as well as the *Tmatches* and *SuccCmapIndex* data structures. These structures are in the same state in which they were at the end of Algorithm Pseudosim, namely at the end of Alg. 6. First, we check whether the input pseudosimulation is unambiguous (line 1). If so, we return all its MTFs (these coincide with simulation preorders). If not, we *disambiguate*

[24] Per Definition 3.5.8.

[25] Since we not only need to find maximal transitive *subsets* of $\lceil\sqsubseteq\rceil$, we also need to ensure they are *simulations* (refer to Def. 3.5.8).

Algorithm 7: `ExampleSearch`
(Illustrative search for simulation preorders in a pseudosimulation)

Input: Pseudosimulation $\lceil\sqsubseteq\rceil$; initialized *Tmatches* and *SuccCmapIndex* from Algorithm Pseudosim; CA A
Output: Consistent simulations of A

```
1  if ⌈⊑⌉ is unambiguous then
2  |  return all maximal transitive fragments[24] of ⌈⊑⌉;
3  consistentSimulations := {};
4  for each (q, s), s.t. |⌈⊑⌉(q, s)| > 1 do
5  |  for γ_q^s ∈ ⌈⊑⌉(q, s) do
6  |  |  CmapRemoveQueue := ⌈⊑⌉(q, s) \ {γ_q^s};
7  |  |  Γ' := ⌈⊑⌉;
8  |  |  Tmatches' := Tmatches;
9  |  |  SuccCmapIndex' := SuccCmapIndex;
10 |  |  perform lines 3-17 of Algorithm 6 with Γ', Tmatches', SuccCmapIndex'
   |  |    until convergence; modifying the three structures in the process;
11 |  |  // Γ' may still be ambiguous – recursively invoke backtracking search
12 |  |  newSimulations := ExampleSearch(Γ', Tmatches', SuccCmapIndex');
13 |  |  consistentSimulations := consistentSimulations ∪ newSimulations;
14 // return only maximal consistent simulations
15 return { ⊑ ∈ consistentSimulations | ∄ ⊑' ∈ consistentSimulations: ⊑ ⊂ ⊑' };
```

$\lceil\sqsubseteq\rceil$ recursively. First, we choose a pair of states (q, s) which has multiple mappings, and for this pair, we choose a single mapping (lines 4 and 5). We remove all other mappings of (q, s) than the selected one (line 6). Then we propagate this change by inductively removing all mappings which depend on the removed mappings, and so forth, until convergence (line 10). We do this by executing the main loop of Alg. 6 with the primed variables (e.g. with Γ' instead of Γ). This modifies the primed variables (that is why we use the primed copies – to not modify the original ones). The result is a new pseudosimulation Γ', which may still be ambiguous. We therefore invoke the search recursively, and all simulations found by the recursive call are added to the set of found consistent simulations (lines 12, 13). We then proceed to the next choice (e.g. next mapping of (q, s)) and repeat the above process. Once we have exhausted all choices, we have found all simulations in $\lceil\sqsubseteq\rceil$ which are unambiguous and transitive (and naturally also reflexive). However, it may happen that some of them are subsets of others. We thus return only the maximal among them (line 15).

5.2.1 Reduction to SAT

Algorithm 7 would be rather difficult to implement, especially considering the search for maximal transitive fragments. As previously mentioned, this problem is NP-hard. We do not expect to be facing this problem often, however. Neither do we expect to frequently encounter ambiguous pseudosimulations. Due to this, we do not need the search for simulation preorders to be particularly efficient. We will thus not use Algorithm 7 or a variation thereof; we will use a more straightforward solution.

The solution relies on reduction to the *Boolean satisfiability problem* (SAT). This problem is also known to be NP-hard [16], but is solved frequently, due to which there exist very efficient solvers for this problem [44; 23]. Although a specialized solver for our particular problem may be more efficient than a reduction to SAT, the inefficiency of the reduction approach is compensated by its simplicity. All that needs to be done is to describe a consistent simulation using *propositional formulae* over *Boolean variables*, based on the computed pseudosimulation $\lceil\sqsubseteq\rceil$ and the transition matches *Tmatches*. The Boolean variables and formulae form a *SAT instance*. We then use an existing SAT solver to find *satisfying models* (variable assignments under which all formulae hold) for this instance. Each of these models corresponds to a consistent simulation. If we find all models, we find all consistent simulations. We construct the SAT instance followingly.

SAT formulation of consistent simulations. We now present the *SAT formulation* of finding consistent simulations, that is, the reduction of the problem of finding consistent simulations to the SAT problem. The input to this reduction is the pseudosimulation $\lceil\sqsubseteq\rceil$; and the *Tmatches* structure, which is in the same state as at the end of Algorithm Pseudosim. The output is an instance of the SAT problem which encodes all consistent simulations in $\lceil\sqsubseteq\rceil$. In the SAT instance, we denote by `Vars` the set of all variables and by Φ the set of all formulae.

1. **Variables.**

   First, we need to define what the variables are. This is in fact quite simple – each mapping γ_q^s between two states corresponds to a Boolean variable $\texttt{var}(\gamma_q^s)$. We note again that the states q and s specified in γ_q^s are important, since we need to distinguish equal mappings which differ only by the states. If the variable $\texttt{var}(\gamma_q^s)$ is valued `true`, then the mapping γ_q^s exists in the simulation (meaning $q \sqsubseteq_{\gamma_q^s} s$ holds), otherwise it does not. By encoding mappings as Boolean variables, we let the SAT solver do the work of disambiguating and finding transitive fragments, which in Alg. 7 we had to do "manually". The set of all variables is $\texttt{Vars} = \{\texttt{var}(\gamma_q^s) \mid \exists\, q, s \in Q\colon \gamma_q^s \in \lceil\sqsubseteq\rceil(q,s)\}$.

2. **Formulae.**

   We must encode the following constraints, which are placed on a consistent simulation, using formulae of propositional logic.

   (a) *Unambiguity* – at most one mapping must exist between every state pair (q, s); even if $q = s$. That is, existence of a mapping γ_q^s from s to q implies non-existence of any other mapping from s to q. Thus Φ contains the following formula for all $q, s \in Q$ and $\gamma_q^s \in \lceil\sqsubseteq\rceil(q,s)$:

   $$\texttt{var}(\gamma_q^s) \Rightarrow \neg \bigvee_{\widehat{\gamma_q^s} \in X} \texttt{var}(\widehat{\gamma_q^s}); \quad \text{where } X = \lceil\sqsubseteq\rceil(q,s) \setminus \{\gamma_q^s\}.$$

(b) *Reflexivity* – each state must simulate itself under the identity mapping. Recall $\lceil\sqsubseteq\rceil$ is reflexive and only identity mappings are allowed from a state to itself. This means that for any $q \in Q$, $\lceil\sqsubseteq\rceil(q,q)$ contains precisely the identity mapping from q to q, denoted ${\gamma_{\text{ID}}}^{(q)}$. Thus Φ contains the following formula for each $q \in Q$ and ${\gamma_{\text{ID}}}^{(q)} \in \lceil\sqsubseteq\rceil(q,q)$:

$$\texttt{var(}{\gamma_{\text{ID}}}^{(q)}\texttt{)}.$$

It is of course unnecessary to have such variables which are always true, but it makes the reduction more straightforward and apparent.

(c) *Transitivity* – if $q \sqsubseteq_{\gamma_1} r$ and $r \sqsubseteq_{\gamma_2} s$, then $q \sqsubseteq_{\gamma_3} s$ where $\gamma_3 = \gamma_1 \circ \gamma_2$. However, the mapping γ_3 may not exist from s to q in $\lceil\sqsubseteq\rceil$. Then it is not possible for $q \sqsubseteq_{\gamma_1} r$ and $r \sqsubseteq_{\gamma_2} s$ to hold simultaneously. We need to consider that in the SAT formulation. Transitivity constraints are quite complex, hence we present them in the form of the following pseudocode. It modifies the set Φ by adding formulae of transitivity constraints:

for $q, r, s \in Q^3$ **do**
  for $\gamma_q^r, \gamma_r^s \in \lceil\sqsubseteq\rceil(q,r) \times \lceil\sqsubseteq\rceil(r,s)$ **do**
    $\gamma_q^s := \gamma_q^r \circ \gamma_r^s$;
    if $\gamma_q^s \in \lceil\sqsubseteq\rceil(q,s)$ **then**
      // composed mapping exists – ensure transitivity
      $\phi :=$ "$\texttt{var(}\gamma_q^r\texttt{)} \wedge \texttt{var(}\gamma_r^s\texttt{)} \Rightarrow \texttt{var(}\gamma_q^s\texttt{)}$";
    else
      // composed mapping does not exist – transitivity not possible
      $\phi :=$ "$\neg\ (\texttt{var(}\gamma_q^r\texttt{)} \wedge \texttt{var(}\gamma_r^s\texttt{))}$";
    $\Phi := \Phi \cup \{\phi\}$;

(d) *Simulation* – each transition τ_q from the weaker state q is simulated by at least one transition τ_s from the stronger state s, under at least one successor mapping $\gamma_{q'}^{s'}$. For all $q, s \in Q$ and all $\gamma_q^s \in \lceil\sqsubseteq\rceil(q,s)$, the set Φ must contain the following formula:

$$\texttt{var(}\gamma_q^s\texttt{)} \implies \bigwedge_{\tau_q = q\text{-}\{\ldots\}\to q'} \left(\bigvee_{(\tau_s, \gamma_{q'}^{s'}) \in Tmatches[\gamma_q^s, \tau_q]} \texttt{var(}\gamma_{q'}^{s'}\texttt{)} \right).$$

That is, if there is a mapping γ_q^s from s to q (meaning s simulates q under γ_q^s), then each outgoing transition τ_q from q has at least one *match*. This match is a simulating transition τ_s from s and a successor mapping $\gamma_{q'}^{s'}$. Therefore, if there is at least one match, then there must be at least one successor mapping $\gamma_{q'}^{s'}$.

Finding simulation preorders via the SAT formulation. The SAT formulation presented above represents a *consistent simulation*. Not all consistent simulations are *simulation preorders*, only the maximal ones. To find all simulation preorders, we may proceed as follows: find all consistent simulations, and keep only the maximal ones.

Finding preorders for *optimal* equivalences. The number of consistent simulations of a CA may be very large in some cases, especially when there are many "equal" counters (with equal lower and upper bounds). Then, the above approach of finding *all* consistent simulations is too impractical. Even the number of simulation preorders may be too large in such cases.

Furthermore, our ultimate goal is to reduce the automaton by a *simulation equivalence*, and preferably, by an *optimal* one. Recall that a simulation equivalence is optimal iff it allows as good a reduction as any other simulation equivalence. Hence we do not really need to find all simulation preorders, we only need to find those, which result in an *optimal simulation equivalence*. For an example, refer to proof of Lemma 3.5.8, case *Maximal* $\not\Rightarrow$ *optimal*. There, one of the simulation preorders leads to the equivalence $\{\{p, q\}\}$, which is not optimal. In that example, the simulation preorder which does lead to the optimal simulation equivalence is the *largest simulation preorder*. For a counterexample where this is not the case (i.e., where the largest simulation preorder does *not* lead to an optimal equivalence), refer again to the same proof, case *Largest*.

We expect that in most CA, any *largest* simulation preorder leads to an optimal equivalence. Even if it does not, the equivalence obtained from it should be comparable with the optimal ones. We therefore propose the following simplification:

Find only a single *largest* consistent simulation,
as it likely provides an optimal simulation equivalence.

This approach is also motivated by the ability of SAT solvers (such as [44]) to efficiently find models which maximize the number of `true`-valued variables.

End of Algorithm Search. Having created a SAT instance and found its model, we then need to convert the model to a simulation $\sqsubseteq$. Let $v(\texttt{var}(\gamma_q^s))$ denote the valuation of $\texttt{var}(\gamma_q^s)$ in the found model. We construct $\sqsubseteq$ by taking all `true`-valued variables: $\sqsubseteq = \{\,(q, s, \gamma_q^s) \in \lceil\sqsubseteq\rceil \mid v(\texttt{var}(\gamma_q^s)) = \texttt{true}\,\}$. Now, we can use the simulation $\sqsubseteq$ for reduction, as described in Chapter 4.

Chapter 6

Experimental Evaluation

We performed an experimental evaluation of the algorithms presented in Chapter 5, using CA obtained from real-world regexes. In this chapter, we first briefly discuss the evaluation environment (the implementation and used hardware); then we describe the input data and how they were processed; lastly, we discuss the performed experiments and obtained results.

6.1 Environment

Implementation. We implemented the proposed Algorithms Pseudosim and Search in the C# language, within the CountingAutomata library [54]. This library itself builds upon the Symbolic Automata Toolkit [57], which it extends with an implementation of counting automata (mostly in accordance with [55]). It allows compilation of regexes to CA, which we used for the evaluation.

In Algorithm Pseudosim, we do not use the INY algorithm for computation of classical simulation on the structural FA A'; we use the novel NoCount algorithm [30] instead. The main reason is its superior memory efficiency: on large automata (e.g. over 2,000 states), the memory consumption of INY has proven prohibitive. Moreover, as a result of using NoCount, the running time on large inputs was improved, whereas on small inputs it remains comparable. We use NoCount instead of other simulation algorithms because there is an existing implementation [49] within the Symbolic Automata Toolkit which is easily modified to suit our purposes.

In Algorithm Search, the used SAT solver is the Z3 Theorem Prover [44]. We opted for this solver due to its maturity, good efficiency, and ease of integration with our implementation. Furthermore, it is able to maximize the number of `true`-valued variables when searching for a satisfying model, which allows us to efficiently find a largest simulation preorder (as discussed at the end of Sec. 5.2.1).

The C# source code was compiled with Microsoft Visual C# Compiler version 4.8.4084.0, for x86-64 CPU, with optimizations enabled.

Hardware and OS. The experiments were performed on a PC with Lenovo 20LJS2EV0R motherboard, Intel Core i5-8350U CPU (with base speed 1.90GHz and peak speed 2.9GHz), equipped with 16GB of RAM; running on OS Windows 10 Version 2004, under the .NET framework version 4.0.30319.

6.2 Input Data

For the evaluation, we used regexes from the database of real-world regexes collected from over 190,000 software projects [20].[26] The initial set contained over 500,000 regexes. From these, we removed those which could not be compiled; the main reasons were:

- the regex used features not supported by the Symbolic Automata Toolkit (such as word boundaries);
- the structure of the regex was not compatible with the CountingAutomata library (e.g. it does not support nested counting loops);
- the compilation time exceeded the allowed maximum of 120 seconds – this was possibly caused by an error in the compilation, as we witnessed excessive memory usage in such cases (all of the available RAM was used by the CA compilation process).

We further removed those regexes which resulted in CA without any counters. The remaining 28,700 regexes were used for evaluation.

When compiling CA into regexes, we used the following options of the CountingAutomata library functions:

- `RegexOptions.Singleline`,
- `keepAnchors=false, unwindLowerBounds=false`,
- `makeMonadic=false`.

6.3 Experiments

We performed three experiments. In the first one, we ran the (bi)simulation algorithm on each of the input regexes individually. Due to the method of RE-to-CA construction used in the CountingAutomata library, the compiled automata did not exhibit much redundancy, as we discuss later. For a better assessment of the capabilities of our reduction algorithm, we performed two additional experiments involving disjunctions of a large number of regexes to artificially increase the CA redundancy.

For each CA, we computed a single largest consistent simulation and largest consistent bisimulation (hereafter referred to as Sim and Bisim respectively). For Sim and Bisim we calculated the *absolute reduction*, and the *percentual (relative) reduction* (also referred to simply as *reduction* and denoted $|\textsc{Sim}|_{\%}$ or $|\textsc{Bisim}|_{\%}$). Absolute reduction is the number of states removed by reducing the CA by the given Bisim, or by the simulation equivalence obtained from the given Sim. Percentual reduction is absolute reduction divided by the number of CA states, in percent. Clearly, $|\textsc{Sim}|_{\%} > |\textsc{Bisim}|_{\%}$, since every bisimulation is also a strongly symmetric simulation. We did not perform the actual reduction of CA by the computed Sim and Bisim, as the number of removed states can be precisely calculated from Sim and Bisim alone.

[26] Available at
https://github.com/ituronova/benchmarks-ca/raw/master/Uniq/uniq-regexes-8.modified.txt
(accessed 27 April 2021).

6.3.1 Experiment 1: Individual Regexes

In this experiment, we computed Sim and Bisim for each regex individually. The characteristics of the obtained CA are listed in Table 6.1a.

The mean percentual reduction was 2.1% for Sim and 1.4% for Bisim; median was 0% for both. This is caused by the following two facts. Firstly, the regexes we used are handmade and small. Such regexes often exhibit less redundancy than e.g. machine-generated regexes. Secondly, the CountingAutomata library implements construction of CA via the so-called *generalized Antimirov derivatives* [55], which produces automata with very little redundancy. As our experiments show, there is indeed not much room for reduction. Later, we will present additional experiments targeting CA with higher redundancy.

Of the 28,700 CA in total, 24,557 (86%) have zero reduction (via Sim or Bisim). Let us now discuss the remaining 4,143 CA, in which $|\text{Sim}|_{\%} > 0\%$. Their characteristics are listed in Table 6.1b. The mean Sim reduction is 14.6% (median 12.5%; maximum 82%). Mean Bisim reduction including values $|\text{Bisim}|_{\%} = 0\%$ is 10.0% (median 7.7%); excluding values $|\text{Bisim}|_{\%} = 0\%$, the mean is 13.8% (median 11.8%). Maximum $|\text{Bisim}|_{\%}$ is 52%. On average, Bisim reduction is 31.2% weaker than Sim reduction (median is 0% weaker).[27]

Figure 6.1a shows the comparison of Sim and Bisim reduction. Plotted are values for the 4,143 CA which have non-zero Sim reduction. The red background color (*heatmap*) indicates the density of values; viz. the more values there are at certain coordinates, the more intense the color. Zero values of the Bisim reduction are plotted at the bottom of the graph, around the value 0.1 on the y-axis.[28]

Of the 4,143 CA with non-zero Sim reduction, 2,671 have $|\text{Sim}|_{\%} = |\text{Bisim}|_{\%}$. From the remaining 1,472 which have $|\text{Sim}|_{\%} > |\text{Bisim}|_{\%}$, 1,124 have $|\text{Bisim}|_{\%} = 0\%$ and the remaining 348 have Bisim reduction 48.2% weaker on average than Sim reduction (median 50.0%). Notably, a majority of CA in this experiment have either zero Sim, or zero Bisim, or Sim equal to Bisim (as is evident from Fig. 6.1a). This is primarily because the reduction mostly removes only 1 state; namely in 79% of the 4,143 non-zero Sims, and 87% of the 3,018 non-zero Bisims. Unsurprisingly, the second most common absolute reduction is 2 states, and so on.

Figure 6.1b shows the absolute Sim reduction with respect to the number of states of the CA. Plotted are again only values with $|\text{Sim}|_{\%} > 0$. The most frequent number of removed states is 1, for every number of CA states up to 52 (inclusive). This further confirms that the CA produced by the CountingAutomata library are quite reduced by construction already.

Table 6.1: Statistics for CA compiled from individual regexes.

(a) Statistics for all CA.

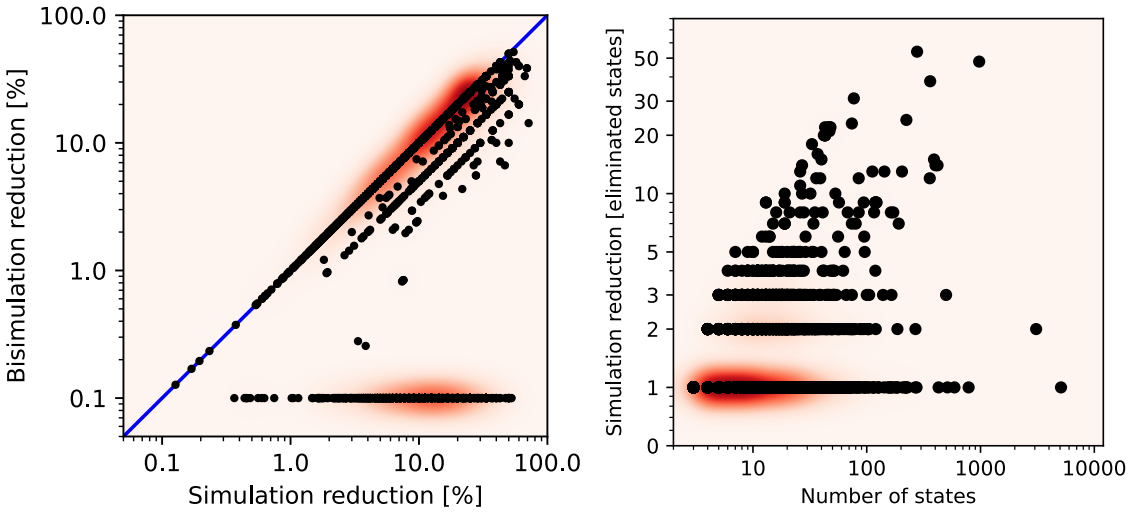
	States	Transitions	Counters
Min	1	1	1
Max	6,605	41,536	248
Mean	18.1	53.9	1.8
Median	7	22	1
Mode	3	10	1

(b) Statistics for CA with $|\text{Sim}|_{\%} > 0$.

	States	Transitions	Counters
Min	3	7	1
Max	5,097	41,536	248
Mean	19.8	108	1.6
Median	10	45	1
Mode	4	15	1

[27] Meaning the reduction obtained by Bisim is $|\text{Bisim}|_{\%} = (100\% - 31.2\%) * |\text{Sim}|_{\%}$.

[28] This is due to technical limitations of the software used to produce the figures.



(a) Comparison of percentual simulation and bisimulation reduction. Values where $|\textsc{Sim}|_{\%} = 0$ are omitted; values where $|\textsc{Bisim}|_{\%} = 0$ are plotted at the bottom (around the value 0.1 on the y-axis).

(b) Absolute simulation reduction as a function of the number of states. The most common case is 4-state CA with 1-state reduction.

Figure 6.1: Results of Experiment 1, concerning individual regexes.

Additional figures. Appendix D presents additional figures for Experiment 1, which show in more detail the dependence of simulation reduction (absolute and relative) on the number of states.

6.3.2 Experiments 2 and 3: Disjunctions of Regexes

In our attempts to achieve a better reduction, we performed two additional experiments, in which we used *disjunctions* of several regexes (we connected the regexes by the | operator, described in Sec. 2.4.2). In each experiment, we selected 8,000 regexes from the original set of 28,700. The 8,000 regexes were then split into 80 batches of 100 regexes each. For each batch, we computed Sim and Bisim on the disjunction of all regexes in the batch.

We attempted to make the batches as large as possible; the size of 100 regexes per batch was chosen since larger batches were resulting in too many errors or timeouts during compilation. In particular, we frequently witnessed excessive memory and time consumption during RE-to-CA compilation; hence we limited the running time per each disjunction to 300 seconds for compilation and 900 seconds for computing simulation and bisimulation. We also encountered some errors during compilation which were not present in Experiment 1. These were caused by an unsupported combination of features present in the individual regexes.[29]

Additional figures. Appendix D presents an additional figure for Experiments 2 and 3, which shows the dependence of simulation running time on the number of states.

[29]Specifically, the Symbolic Automata Toolkit does not support combination of *greedy* and *non-greedy* matching.

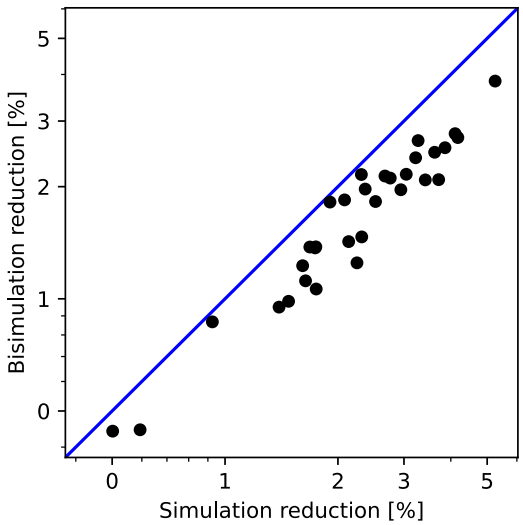
Experiment 2: Disjunctions of Random Regexes

In this experiment, the 8,000 regexes were chosen randomly, and were randomly split into batches of 100. Each batch thus contained regexes which were "independent" (as opposed to Experiment 3). Of the 80 disjunctions, 47 did not pass compilation (ended with an error or a timeout after 300s) and 1 timed out on computing Sim (after 900s). The characteristics of CA obtained from the remaining 32 disjunctions are listed in Table 6.2a. The mean running time of the simulation algorithm was 47.5 seconds; median was 20 seconds.

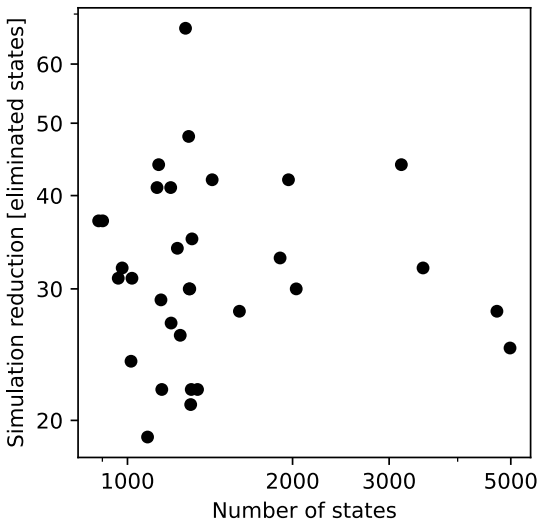
Figure 6.2a shows the comparison of Sim and Bisim reduction on the 32 successfully evaluated CA. All 32 CA have non-zero Sim (min. 0.5%) *and* Bisim (min. 0.4%). Furthermore, Bisim reduction is *always* weaker than Sim, on average by 26.2% (median 27.2%). The reason is much larger complexity of the analyzed CA, compared to Exp. 1. In this experiment, the CA allow more room for difference between Sim and Bisim, whereas in Exp. 1, both Sim and Bisim absolute reductions were mostly either 0 or 1 state.

The overall relative reduction is significantly worse than in Experiment 1. Sim reduction achieves a maximum of 5.3% (cf. 82% in Exp. 1); mean 2.5% (vs. 14.6%), median 2.3% (vs. 12.5%). Bisim reduction is – naturally – no better, with max. 3.8%, mean 1.8% and median 1.8%. This is primarily caused by the fact that in Exp. 1, most CA had few states (median was 10 states per CA), so the percentual reduction was quite high even with a very modest absolute reduction. In this experiment, on the other hand, the CA are obviously larger, with a median of 1,284.5 states per CA. Also, combining random, dissimilar regexes did not particularly increase redundancy of the obtained automata, and this again does not help improve reduction. Therefore, in Experiment 3 we combined regexes which are more similar and obtained better results.

Figure 6.2b shows the dependence of absolute reduction on the number of states. Observe that the number of reduced states does not seem to be particularly increasing with the increasing number of CA states.



(a) Comparison of percentual simulation and bisimulation reduction.



(b) Absolute simulation reduction as a function of the number of states.

Figure 6.2: Results of Experiment 2, concerning disjunctions of random regexes.

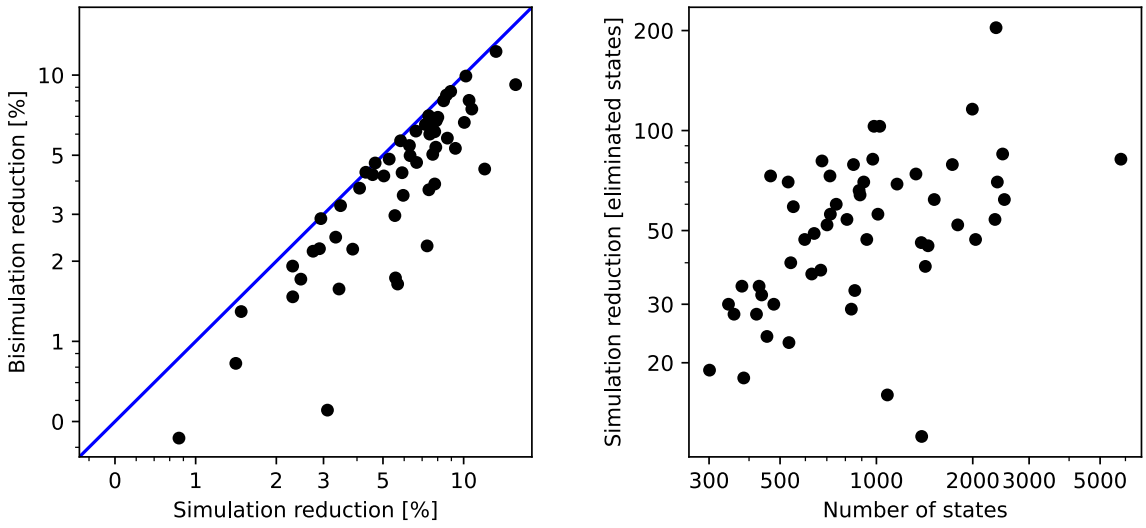
Experiment 3: Disjunctions of Similar Regexes

In this experiment, we started with a lexicographically sorted sequence of the original 28,700 regexes, and selected a subsequence of 8,000 successive regexes, These were then split into 80 successive batches; each batch was kept sorted. Each batch thus contained regexes which were "similar" (as opposed to Experiment 2), since they were lexicographically neighboring. Of the 80 disjunctions, 23 did not pass compilation (ended with an error or a timeout after 300s) and 3 timed out on computing Sim (after 900s). The characteristics of CA obtained from the remaining 54 disjunctions are listed in Table 6.2b. The mean running time of the simulation algorithm was 44.2 seconds; median was 10.5 seconds.

Figure 6.3a shows the comparison of Sim and Bisim reduction on the 54 successfully evaluated CA. Yet again, all 54 CA have non-zero Sim (min. 0.9%) *and* Bisim (min. 0.4%). Bisim reduction is weaker than Sim in all except 2 cases; on average (incl. the 2 cases) by 26.7% (median 22.2%). The difference between Sim and Bisim is comparable to that observed in Exp. 2.

The overall relative reduction is now still worse than in Experiment 1, but considerably better than in Exp. 2. Sim reduction achieves a maximum of 15.6% (cf. 5.3% in Exp. 2); mean 6.3% (vs. 2.5%), median 6.3% (vs. 2.3%). Bisim reduction is comparable, with max. 12.3%, mean 4.7% and median 4.6%. The improvement against Exp. 2 is caused by the increased similarity of regexes in disjunction, and as a result, increased redundancy of the obtained automata.

Figure 6.3b shows the dependence of absolute reduction on the number of states. Interestingly, the number of reduced states is clearly increasing with the increasing total number of CA states. This is unlike in Experiment 2, where such tendency was not present. Again, it is caused by the redundancy of the CA – the higher the total number of states, the higher the number of redundant states, since the regexes in disjunction are largely similar.



(a) Comparison of percentual simulation and bisimulation reduction.

(b) Absolute simulation reduction as a function of the number of states.

Figure 6.3: Results of Experiment 3, concerning disjunctions of similar regexes.

Table 6.2: Statistics for CA compiled from disjunctions of regexes.

(a) Statistics for Exp. 2 (unrelated CA).

	States	Transitions	Counters
Min	886	23,081	138
Max	4,984	34,467	217
Mean	1,624.3	28,733.8	176.1
Median	1,284.5	29,112	176
Mode	1,898	30,951	177

(b) Statistics for Exp. 3 (similar CA).

	States	Transitions	Counters
Min	301	11,263	89
Max	5,807	67,923	390
Mean	1,129.7	32,114.2	154.4
Median	856	30,735	144.0
Mode	1,011	46,967	127

6.4 Discussion of Results

Here, we would like to make a few remarks about the preformed experimental evaluation and the obtained results.

Reduction efficiency of the proposed algorithms. We extensively assessed the reduction capabilities of our algorithm on CA constructed from real-world regexes. Albeit the achieved results do not seem remarkable at a glance, we consider them satisfactory at the very least, considering the circumstances. Firstly, the used regexes are hand-made and mostly small, and such regexes often result in automata with little redundancy (even in the classical FA case). Furthermore, the RE-to-CA construction implemented by the CountingAutomata library uses generalized Antimirov derivatives, which are known for producing nearly optimal automata. Due to this, the CA we used are not particularly suitable for assessing the efficiency of our reduction algorithm. However, since the CountingAutomata library is the only implementation of CA known to us, we had no other choice than to perform the evaluation using this library.

In other areas than regex matching, the proposed algorithm may prove much more useful. For example, the area of formal verification has a large potential for application of counting automata. In this area, automata are used extensively, and frequently are machine-made and exhibit much larger redundancy than those from regexes. In particular, redundant automata are often encountered in regular model checking [7], where the reduction of the used automata can be decisive for the overall performance of the algorithm [6]. Another application is translation of logic formulae to CA [50] and subsequent manipulation, which could also benefit from size reduction.

Comparison with other methods. In an evaluation like this, one would expect to see a comparison between the proposed and existing methods. However, since no methods for reduction of (or even computing simulation in) CA are known to us, possibilities of such comparisons are highly limited. To alleviate this deficiency, we propose the following comparison method relying on classical FA simulation. Unfortunately, we were not able to implement and use this comparison method, but we may do so in future research.

The proposed method utilizes the configuration CA $SFA(A)$ of an input CA A. We reduce A by the proposed simulation-based reduction to obtain A', and from A', we obtain the configuration automaton $SFA(A')$. We then reduce the *original* configuration automaton $SFA(A)$ by classical means to obtain a reduced FA $SFA(A)'$. Finally, we compare the sizes of $SFA(A')$ and $SFA(A)'$ to obtain relative efficiency of the proposed reduction method on the input CA A.

Chapter 7

Conclusions and Future Work

In this work, we studied simulation on classical finite automata (FA) for the purpose of size reduction. The aim of this work was to extend simulation-based reduction techniques to *counting automata (CA)* introduced in [55]. The extension to CA is nontrivial, as the CA formalism is much more complex than that of FA. Furthermore, to to our knowledge, such adoption of reduction methods to CA or similar models has not been attempted before.

Our first main contribution is the novel notion of *parameterized simulation* on counting automata. We defined parameterized simulation, providing a rationale for the chosen definition. In an analysis of its properties, we showed that a counting automaton can have several simulation preorders, whereas a classical finite automaton has precisely one. Moreover, we showed that a CA cannot be reduced by an arbitrary simulation; this simulation has to satisfy certain conditions. This motivated us to develop the notion of *consistent simulation*, which is a simulation that can be used for reduction.

We then proposed means of reducing a CA by a consistent simulation relation, and gave a sketch of proof showing that this reduction preserves the language of the CA. Since the proposed reduction cannot be applied under the original formalism of CA (as presented in [55]), we extended this formalism by a new *counter rename* operation on transitions.

The second main contribution of this thesis is the algorithm for computing simulations in CA. It consists of two sub-algorithms – Algorithm Pseudosim and Algorithm Search. Algorithm Pseudosim finds a *pseudosimulation*, which is a structure containing all the simulation preorders of the input CA. It employs various pruning and optimization methods, which improve its performance and thus its potential of practical use. Algorithm Search is then used to search for preorders in the pseudosimulation. These algorithms can also be modified to compute the bisimulation instead of simulation.

We implemented the proposed algorithms and performed an extensive experimental evaluation. The evaluation used over 28,000 real-world regular expressions (RE) with counting, collected from thousands of software projects. These regexes were compiled into counting automata, for which we computed the simulation and bisimulation using the implemented algorithms. Unfortunately, we were not able to fully assess the reduction efficiency of our algorithms, as the used implementation of RE-to-CA construction produces automata which are nearly optimal. For this reason, we performed additional evaluation on specially crafted automata with higher redundancy. The results – under the circumstances – are promising, and show the viability of the proposed reduction methods.

Future work. This work has opened numerous problems and opportunities yet to be addressed. The first one is the correctness of the proposed reduction methods. We would like to obtain a formal proof of the correctness, or at least a proof for some (commonly occurring) restricted case. This would involve full proofs of correctness of the simulation relation itself (i.e., that it implies language inclusion) as well as the merging method (i.e., that it preserves language of the automaton), perhaps based on the sketches presented in this work.

The second desirable extension to this work is an experimental evaluation on other automata than those obtained from regexes via the generalized Antimirov derivative construction of [55]. This would allow us to better assess the reduction capabilities of our algorithm, as well as its performance. Additionally, it would be interesting to compare the reduction efficiency of our CA reduction algorithm to that of classical FA simulation-reduction. This involves comparing two FA – one obtained by first reducing the CA by our algorithm and then converting it to FA; one obtained by converting the CA to FA and then reducing it by classical simulation. The closer the former gets to the latter, the better the efficiency of our proposed reduction.

The third problem is the extension of the proposed methods to other models of automata similar to CA. One such model is their deterministic counterpart, *counting-set automata* (also presented in [55]). Other models are variations of CA arising from different applications than regexes; for example, verification and decision procedures of logics. In particular, the work [50] presents an algorithm for translation of formulae of linear temporal logic [45] with bounded repetition to a variation of CA based on Büchi automata [8]. This model of automata is slightly different from the CA considered in this work, although the adoption of the proposed method seems very much possible. Besides the existing work [50], there is a great potential for application of counting automata in verification, especially in the areas of string solving (e.g. [2; 14]) and regular model checking [7]. In these areas, nondeterministic automata with high redundancy arise often, and many algorithms rely on their reduction to achieve competitive performance. This necessitates efficient means of size reduction for counting automata, such as presented in this work.

Bibliography

[1] Abdulla, P. A., Atig, M. F., Chen, Y.-F., Diep, B. P., Holík, L. et al. Flatten and Conquer: A Framework for Efficient Analysis of String Constraints. In: Cohenm, A. and Vechev, M., ed. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2017, p. 602–617. PLDI 2017. DOI: 10.1145/3062341.3062384. ISBN 9781450349888.

[2] Abdulla, P. A., Atig, M. F., Diep, B. P., Holík, L. and Janků, P. Chain-Free String Constraints. In: Chen, Y.-F., Cheng, C.-H. and Esparza, J., ed. *Automated Technology for Verification and Analysis*. Cham, Switzerland: Springer International Publishing, 2019, p. 277–293. ISBN 978-3-030-31784-3.

[3] Abdulla, P. A., Chen, Y.-F., Holík, L., Mayr, R. and Vojnar, T. When Simulation Meets Antichains. In: Esparza Estaun, F. J. and Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2010, p. 158–174. ISBN 978-3-642-12002-2.

[4] Arnold, A., Dicky, A. and Nivat, M. A note about minimal non-deterministic automata. *Bulletin of the EATCS*. European Association for Theoretical Computer Science. June 1992, no. 47, p. 166–169.

[5] Boigelot, B., Jodogne, S. and Wolper, P. An Effective Decision Procedure for Linear Arithmetic over the Integers and Reals. *ACM Trans. Comput. Logic*. New York, NY, USA: Association for Computing Machinery. July 2005, vol. 6, no. 3, p. 614–633. DOI: 10.1145/1071596.1071601. ISSN 1529-3785.

[6] Bouajjani, A., Habermehl, P., Holík, L., Touili, T. and Vojnar, T. Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: Ibarra, O. H. and Ravikumar, B., ed. *Implementation and Applications of Automata*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2008, p. 57–67. ISBN 978-3-540-70844-5.

[7] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*. April 2012, vol. 14, no. 2, p. 167–191. DOI: 10.1007/s10009-011-0205-y. ISSN 1433-2787.

[8] Büchi, J. R. On a Decision Method in Restricted Second Order Arithmetic. In: Mac Lane, S. and Siefkes, D., ed. *The Collected Works of J. Richard Büchi*. New

York, NY: Springer New York, 1990, p. 425–435. DOI: 10.1007/978-1-4613-8928-6_23. ISBN 978-1-4613-8928-6.

[9] Bustan, D. and Grumberg, O. Simulation-Based Minimization. *ACM Trans. Comput. Logic*. New York, NY, USA: Association for Computing Machinery. 2003, vol. 4, no. 2, p. 181–206. DOI: 10.1145/635499.635502. ISSN 1529-3785.

[10] Chakraborty, S., Ghosh, S., Jha, N. and Roy, S. Maximal and Maximum Transitive Relation Contained in a Given Binary Relation. In: Xu, D., Du, D. and Du, D., ed. *Computing and Combinatorics*. Cham, Switzerland: Springer International Publishing, 2015, p. 587–600. ISBN 978-3-319-21398-9.

[11] Chakraborty, S. and Jha, N. Exact algorithms for maximum transitive subgraph problem. In: Randerath, B., ed. *Proceedings of 15th Cologne-Twente Workshop on Graphs and Combinatorial Optimization, CTW 2017*. Centrum Wiskunde & Informatica, Amsterdam, The Netherlands, 2020, p. 47–51.

[12] Champarnaud, J.-M. and Coulon, F. NFA reduction algorithms by means of regular inequalities. *Theoretical Computer Science*. 2004, vol. 327, no. 3, p. 241 – 253. DOI: 10.1016/j.tcs.2004.02.048. ISSN 0304-3975. Developments in Language Theory.

[13] Champarnaud, J.-M. and Coulon, F. Erratum to "NFA Reduction Algorithms by Means of Regular Inequalities" [Theoret. Comput. Sci. 327(2004) 241-253]. *Theor. Comput. Sci.* Barking, Essex, UK: Elsevier Science Publishers Ltd. November 2005, vol. 347, 1–2, p. 437–440. DOI: 10.1016/j.tcs.2005.07.001. ISSN 0304-3975.

[14] Chen, T., Hague, M., He, J., Hu, D., Lin, A. W. et al. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In: Hung, D. V. and Sokolsky, O., ed. *Automated Technology for Verification and Analysis*. Cham, Switzerland: Springer International Publishing, 2020, p. 325–342. ISBN 978-3-030-59152-6.

[15] Clemente, L. and Mayr, R. Efficient reduction of nondeterministic automata with application to language inclusion testing. *Logical Methods in Computer Science*. February 2019, vol. 15, no. 1. DOI: 10.23638/LMCS-15(1:12)2019.

[16] Cook, S. A. The Complexity of Theorem-Proving Procedures. In: Lewis, P. M., ed. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. STOC '71. DOI: 10.1145/800157.805047. ISBN 9781450374644. Available at: https://doi.org/10.1145/800157.805047.

[17] Cécé, G. Foundation for a series of efficient simulation algorithms. In: Ouaknine, J., ed. *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2017, p. 1–12. DOI: 10.1109/LICS.2017.8005069. ISBN 9781509030194.

[18] D'Antoni, L. and Veanes, M. Minimization of Symbolic Automata. In: Jagannathan, S., ed. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2014, p. 541–553. POPL '14. DOI: 10.1145/2535838.2535849. ISBN 9781450325448.

[19] D'Antoni, L. and Veanes, M. Forward Bisimulations for Nondeterministic Symbolic Finite Automata. In: Legay, A. and Margaria, T., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2017, p. 518–534. ISBN 978-3-662-54577-5.

[20] Davis, J. C. Rethinking Regex Engines to Address ReDoS. In: Dumas, M., Pfahl, D., Apel, S. and Alessandra, R., ed. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019, p. 1256–1258. ESEC/FSE 2019. DOI: 10.1145/3338906.3342509. ISBN 9781450355728.

[21] De Wulf, M., Doyen, L., Henzinger, T. A. and Raskin, J. F. Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T. and Jones, R. B., ed. *Computer Aided Verification*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2006, p. 17–30. ISBN 978-3-540-37411-4.

[22] Eberl, M. *Efficient and Verified Computation of Simulation Relations on NFAs*. Minga, Boarn, 2012. Bachelor's thesis. Technical University of Munich, Department of Informatics. Supervisor Tobias Nipkow.

[23] Eén, N. and Sörensson, N. An Extensible SAT-solver. In: Giunchiglia, E. and Tacchella, A., ed. *Theory and Applications of Satisfiability Testing*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2004, p. 502–518. ISBN 978-3-540-24605-3.

[24] Elgaard, J., Klarlund, N. and Møller, A. MONA 1.x: New techniques for WS1S and WS2S. In: Hu, A. J. and Vardi, M. Y., ed. *Computer Aided Verification*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 1998, p. 516–520. ISBN 978-3-540-69339-0.

[25] Gelade, W., Gyssens, M. and Martens, W. Regular Expressions with Counting: Weak versus Strong Determinism. In: Královič, R. and Niwiński, D., ed. *Mathematical Foundations of Computer Science 2009*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2009, p. 369–381. DOI: 10.1007/978-3-642-03816-7_32. ISBN 978-3-642-03816-7.

[26] Gramlich, G. and Schnitger, G. Minimizing nfa's and regular expressions. *Journal of Computer and System Sciences*. 2007, vol. 73, no. 6, p. 908 – 923. DOI: https://doi.org/10.1016/j.jcss.2006.11.002. ISSN 0022-0000.

[27] Haertel, M. e. a. *GNU grep*. [n.d.]. Online; accessed 18 January 2021. Available at: https://www.gnu.org/software/grep/.

[28] Heizmann, M., Hoenicke, J. and Podelski, A. Software Model Checking for People Who Love Automata. In: Sharygina, N. and Veith, H., ed. *Computer Aided Verification*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2013, p. 36–52. ISBN 978-3-642-39799-8.

[29] Henzinger, M. R., Henzinger, T. A. and Kopke, P. W. Computing Simulations on Finite and Infinite Graphs. In: Raghavan, P., ed. *Proceedings of the 36th Annual*

Symposium on Foundations of Computer Science. Washington, DC, USA: IEEE Computer Society, 1995, p. 453. FOCS '95. ISBN 0818671831.

[30] Holík, L., Lengál, O., Síč, J., Veanes, M. and Vojnar, T. Simulation Algorithms for Symbolic Automata. In: Lahiri, S. K. and Wang, C., ed. *Automated Technology for Verification and Analysis*. Cham, Switzerland: Springer International Publishing, 2018, p. 109–125. ISBN 978-3-030-01090-4.

[31] Holík, L., Lengál, O., Síč, J., Veanes, M. and Vojnar, T. *Simulation Algorithms for Symbolic Automata (Technical Report)*. CoRR, 2018. Available at: https://arxiv.org/abs/1807.08487.

[32] Hopcroft, J. An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Z. and Paz, A., ed. *Theory of Machines and Computations*. Academic Press, 1971, p. 189 – 196. DOI: 10.1016/B978-0-12-417750-5.50022-1. ISBN 978-0-12-417750-5.

[33] Hopcroft, J. E., Motwani, R. and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. 2nd ed. Boston, MA, USA: Addison Wesley, 2001. ISBN 0-201-44124-1.

[34] Hovland, D. Regular Expressions with Numerical Constraints and Automata with Counters. In: Leucker, M. and Morgan, C., ed. *Theoretical Aspects of Computing - ICTAC 2009*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2009, p. 231–245. ISBN 978-3-642-03466-4.

[35] Hovland, D. The Membership Problem for Regular Expressions with Unordered Concatenation and Numerical Constraints. In: Dediu, A.-H. and Martín Vide, C., ed. *Language and Automata Theory and Applications*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2012, p. 313–324. ISBN 978-3-642-28332-1.

[36] Ilie, L., Navarro, G. and Yu, S. On NFA Reductions. In: Karhumäki, J., Maurer, H., Păun, G. and Rozenberg, G., ed. *Theory Is Forever: Essays Dedicated to Arto Salomaa on the Occasion of His 70th Birthday*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2004, p. 112–124. DOI: 10.1007/978-3-540-27812-2_11. ISBN 978-3-540-27812-2.

[37] Ilie, L. and Yu, S. Algorithms for Computing Small NFAs. In: Diks, K. and Rytter, W., ed. *Mathematical Foundations of Computer Science 2002*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2002, p. 328–340. ISBN 978-3-540-45687-2.

[38] Ilie, L. and Yu, S. Reducing NFAs by invariant equivalences. *Theoretical Computer Science*. Barking, Essex, UK: Elsevier Science Publishers Ltd. 2003, vol. 306, no. 1, p. 373 – 390. DOI: 10.1016/S0304-3975(03)00311-6. ISSN 0304-3975.

[39] Jiang, T. and Ravikumar, B. Minimal NFA Problems are Hard. *SIAM Journal on Computing*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics. 1993, vol. 22, no. 6, p. 1117–1141. DOI: 10.1137/0222067.

[40] Kripke, S. A. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*. 1963, vol. 16, no. 1, p. 83–94.

[41] Louden, K. C. *Compiler Construction: Principles and Practice*. 1st ed. Boston, MA, USA: PWS Publishing Co., 1997. ISBN 978-0-534-93972-4.

[42] Milner, R. *Communication and Concurrency*. 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN 978-0-13-115007-2.

[43] Moore, F. R. On the Bounds for State-Set Size in the Proofs of Equivalence Between Deterministic, Nondeterministic, and Two-Way Finite Automata. *IEEE Transactions on Computers*. Los Alamitos, CA, USA: IEEE Computer Society. October 1971, vol. 20, no. 10, p. 1211–1214. DOI: 10.1109/T-C.1971.223108. ISSN 1557-9956.

[44] Moura, L. de and Bjørner, N. Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R. and Rehof, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2008, p. 337–340. ISBN 978-3-540-78800-3.

[45] Pnueli, A. The Temporal Logic of Programs. In: IEEE. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, 1977, p. 46–57. SFCS '77. ISSN 0272-5428.

[46] Rabin, M. O. and Scott, D. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*. 1959, vol. 3, no. 2, p. 114–125. DOI: 10.1147/rd.32.0114.

[47] Ranzato, F. and Tapparo, F. An efficient simulation algorithm based on abstract interpretation. *Information and Computation*. 2010, vol. 208, no. 1, p. 1 – 22. DOI: https://doi.org/10.1016/j.ic.2009.06.002. ISSN 0890-5401.

[48] Sangiorgi, D. On the Origins of Bisimulation and Coinduction. *ACM Transactions on Programming Languages and Systems*. New York, NY, USA: Association for Computing Machinery. May 2009, vol. 31, no. 4. DOI: 10.1145/1516507.1516510. ISSN 0164-0925.

[49] Síč, J. *Symbolicsimulation*. Bitbucket, [n.d.]. Online – Git repository; accessed 5 February 2021 (commit 55d32b4f0c4ecaf06a294aa4e1711d91c8cc09ba). Available at: https://bitbucket.org/jsic/symbolicsimulation.git.

[50] Slezáková, A. *Převod LTL formulí s omezenými operátory do automatů s čítači*. Brno, The Czech Republic, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Lukáš Holík.

[51] Smith, R., Estan, C., Jha, S. and Siahaan, I. Fast Signature Matching Using Extended Finite Automaton (XFA). In: Sekar, R. and Pujari, A. K., ed. *Information Systems Security*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2008, p. 158–172. ISBN 978-3-540-89862-7.

[52] Spencer, H. A Regular-Expression Matcher. In: Schumacher, D., ed. *Software Solutions in C*. San Diego, CA, USA: Academic Press Professional, Inc., 1994, p. 35–71. ISBN 978-0-12-632360-3.

[53] Thompson, K. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. June 1968, vol. 11, no. 6, p. 419–422. DOI: 10.1145/363347.363387. ISSN 0001-0782.

[54] Turoňová, L. et al. *CountingAutomata*. Brno University of Technology, [n.d.]. Online – Git repository; accessed 4 February 2021 (commit c8e881c52959090f38453d0e2fb247ab0abd9309). Available at: https://pajda.fit.vutbr.cz/ituronova/countingautomata.git.

[55] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M. et al. Regex Matching with Counting-Set Automata. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. November 2020, vol. 4, OOPSLA. DOI: 10.1145/3428286.

[56] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M. et al. *Regex Matching with Counting-Set Automata*. MSR-TR-2020-31. Microsoft, September 2020.

[57] Veanes, M. and Bjørner, N. Symbolic Automata: The Toolkit. In: Flanagan, C. and König, B., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2012, p. 472–477. ISBN 978-3-642-28756-5.

[58] Wolper, P. and Boigelot, B. On the Construction of Automata from Linear Arithmetic Constraints. In: Graf, S. and Schwartzbach, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg, 2000, p. 1–19. ISBN 978-3-540-46419-8.

[59] Yannakakis, M. Node-and Edge-Deletion NP-Complete Problems. In: Lipton, R. J., ed. *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1978, p. 253–264. STOC '78. DOI: 10.1145/800133.804355. ISBN 9781450374378.

Appendices

Appendix A

Left Simulation in CA

In this appendix we attend to the *left simulation in CA*, denoted $\sqsubseteq^{(L)}$. We discuss it merely informally, without giving precise definitions. The main reason is that computing left simulation in CA is intrinsically more complex than computing the right simulation. Therefore, our main focus is on the latter; the following discussion of the former is included only for completeness. In this appendix, we assume every counter mapping is total on C. Otherwise, we would need to adapt the definition of counter mappings to left simulation.

Relation of left simulation to left language. Right simulation has no regard for counter memories attainable in the respective weaker and stronger state. This is because for *any* memory $\mathfrak{m}$, the right language of the weaker state is a subset of the right language of the stronger state: $\mathcal{L}(\langle q, \mathfrak{m}\rangle) \subseteq \mathcal{L}(\langle q, \gamma^{-1}(\mathfrak{m})\rangle)$, per Eq. 3.8.

However, when merging left-simulation-equivalent states, we must ensure that all memories attainable in q are also attainable in s. This is to avoid introducing new words into the language of the merged state. For example, assume $w \in \mathcal{L}_R(\langle s, \mathfrak{m}_s\rangle)$ where $\mathfrak{m}_s$ is *not* attainable in s; but $w \notin \mathcal{L}_R(\langle q, \gamma(\mathfrak{m}_s)\rangle)$ where $\gamma(\mathfrak{m}_s)$ *is* attainable in q. Then if we merged q and s, the resulting state would accept the word w with memory $\mathfrak{m}_s$, which was not possible in the original automaton. Therefore, the following must hold for the left simulation:

$$q \sqsubseteq_{\gamma}^{(L)} s \implies \mathfrak{M}(q) \subseteq \{\, \gamma(\mathfrak{m}_s) \mid \mathfrak{m}_s \in \mathfrak{M}(s) \,\}. \tag{A.1}$$

Moreover, each memory $\mathfrak{m}_s$ attainable in the stronger state must be attainable via at least all the words as its corresponding memory $\mathfrak{m}_q = \gamma(\mathfrak{m}_s)$ attainable in the weaker state. This simply means:

$$q \sqsubseteq_{\gamma}^{(L)} s \implies \forall \mathfrak{m}_q \in \mathfrak{M}(q) \colon \mathcal{L}_L(\langle q, \mathfrak{m}_q\rangle) \subseteq \mathcal{L}_L(\langle s, \gamma^{-1}(\mathfrak{m}_q)\rangle). \tag{A.2}$$

Now let us examine what happens with the non-attainable memories. If some $\mathfrak{m}'_q$ is not attainable in q, it means $\mathcal{L}_L(\langle q, \mathfrak{m}'_q\rangle) = \emptyset$. Therefore, $\mathcal{L}_L(\langle q, \mathfrak{m}'_q\rangle) \subseteq \mathcal{L}_L(\langle s, \gamma^{-1}(\mathfrak{m}'_q)\rangle)$ holds (more still, it holds for any s). In fact, we can do the obvious and adopt Equation 3.8 to the left simulation, from which we obtain

$$q \sqsubseteq_{\gamma}^{(L)} s \implies \forall\, \mathfrak{m}_q \in \mathfrak{M} \colon \mathcal{L}_L(\langle q, \mathfrak{m}_q\rangle) \subseteq \mathcal{L}_L(\langle s, \gamma^{-1}(\mathfrak{m}_q)\rangle). \tag{A.3}$$

It is easy to see that Eq. A.3 is equivalent to the conjunction of Eq. A.1 and A.2. Further, it is rather trivially analogous to Equation 3.8 and that gives a strong impression that this implication should indeed hold.

Chapter 4 presents a CA reduction technique based on right-language-equivalence. We presume that the same technique can also be used for reducing a CA by *left-language-equivalence*, and thus (by Eq. A.3) also left-simulation-equivalence. Although possible, such reduction is not practical, as computing the left simulation in CA suffers from a significant difficulty described below. This is also the reason why we do not develop the notion of left simulation any further – it will not be useful anyway.

Computing left simulation. In right simulation, for $q \sqsubseteq_\gamma s$ we assume that the memories $\mathfrak{m}_q$ and $\mathfrak{m}_s$ are equal under the counter mapping, viz. $\mathfrak{m}_q = \gamma(\mathfrak{m}_s)$. After executing a transition τ_q from q and its simulating transition τ_s from s, the respective memories become $\mathfrak{m}'_q$ and $\mathfrak{m}'_s$. The definition of right CA simulation ensures that there exists a successor counter mapping γ' under which the successor memories are again equal, viz. $\mathfrak{m}'_q = \gamma'(\mathfrak{m}'_s)$.

In left simulation, we need to ensure the same for *predecessor memories*. If $\mathfrak{m}_q$ and $\mathfrak{m}_s$ are equal in q and s under γ, then the memories in predecessor states p' and s' need to be equal under some *predecessor mapping* γ'. (We leave out some details but the idea should be clear.)

The essential problem lies in *resetting updates* – EXIT or EXIT1. By executing either of these updates, we lose the previous value of the counter. We then cannot know the value of a counter before a transition (i.e., in the predecessor states) if the counter is reset on the transition. To be accurate, we *could* compute all the possible counter values in the predecessor state. But such computation is much more complex than what we need for right simulation. In left simulation, we would need to perform global static analysis of the entire automaton to obtain the set of possible memories in the predecessor state. In right simulation, on the other hand, to compute the value of a counter after executing a transition, we simply look at the current value of the counter and its update on the transition.

Due to this, we also cannot use the technique presented in section 2.3.3. There, it was shown how we can compute the *left* simulation preorder in FA using the algorithm for the *right* simulation preorder. This is done by using the reverse automaton of a FA. If we were to apply the same principle and reuse the existing algorithm for right CA simulation, we would first need to define a reverse CA. This reverse CA would require *inverse counter updates* on its transitions, instead of the ordinary updates in the original CA. This is again problematic because of the mentioned updates EXIT and EXIT1. Since these operations are not injective, their inverse would be *set-valued*. Then we would need to redefine the entire formalism of counter memories, counting automata, and its semantics. Naturally, all of that would be of no use, since then we could not reuse the existing algorithm.

Appendix B

Upper Bounds of Complexity of Algorithm Pseudosim and Algorithm Search

This appendix presents an analysis of Algorithms Pseudosim and Search. For Algorithm Pseudosim, we derive exact upper bounds of space complexity and approximate upper bounds of time complexity. For Algorithm Search, we derive the upper bounds of number and total size of formulae, and the number of variables.

B.1 Time and Space Complexity of Algorithm Pseudosim

We give a very simple and approximate analysis of Algorithm Pseudosim. We obtain upper bounds of its space complexity, and rough upper bounds of the time complexity. These bounds of time complexity are merely approximate, hence they are not merely non-tight, but likely also inexact. In the following, let $n = |Q|$, $m = |\Delta|$ and $c = |C|$.

Space complexity. We establish the upper bound of the space complexity of Algorithm Pseudosim by examining the used data structures:

1. Γ: we obtain the complexity as the sum of complexities of all Γ^s_q. Each such Γ^s_q contains total injections from C_s to C_q, where $C_s \subseteq C$ and $C_q \subseteq C$. The upper bound for $|\Gamma^s_q|$ is thus the number of total injections from C to C. This is the number of *bijections* on C, which is $c!$. The total space complexity of Γ is then $O(n^2c!)$.

2. *CmapRemoveQueue*: the queue can contain at most all the mappings in Γ. Thus the space complexity is again $O(n^2c!)$.

3. *Tmatches*: each *key* (γ^s_q, τ_q) is associated with a number of *values* $(\tau_s, \gamma^{s'}_{q'})$. There are at most $n^2c! \cdot m$ keys – $n^2c!$ possible γ^s_q and m possible τ_q; and for each key at most $m \cdot c!$ values – m possible τ_s and, since (q', s') are now fixed, $c!$ possible $\gamma^{s'}_{q'}$. The total space complexity is then $O(n^2c! \cdot m \cdot m \cdot c!) = O(n^2m^2c!^2)$.

4. *SuccCmapIndex*: contains precisely the same data as *Tmatches*, only in different format. Space complexity is the same: $O(n^2m^2c!^2)$.

In total, the space complexity is $O(n^2m^2c!^2)$. The given space complexity bound is exact (i.e., correct), albeit possibly not tight.

Time complexity. We show *approximate* upper bounds of time complexity. Although these bounds may be inaccurate, we nonetheless consider them valuable, hence we present them here. We assume operations on queues, sets and associative arrays to have amortized constant time complexity, as well as line 4 of Alg. 4 which checks simulation between two transitions according to the definition of CA simulation.

1. *Initialization phase*: computing the initial $\ddot{\Gamma}$ takes approximately $O(n^2c!)$ time as this is its space complexity (it is the same as of Γ). The actual bound may be higher. The complexity of constructing A', computing its $\preccurlyeq$, and performing pruning by $\preccurlyeq$ is subsumed by the remaining computations. Computing *Tmatches* and *SuccCmapIndex* enumerates first all mappings in $\ddot{\Gamma}$ (there is $O(n^2c!)$ of them), then for each mapping all weaker transitions ($O(m)$), for each weaker transition all stronger transitions ($O(m)$), and for each stronger transition all mappings between the successors ($O(c!)$ of them). In total, it takes time $O(n^2c! \cdot m \cdot m \cdot c!) = O(n^2m^2c!^2)$.

2. *Inductive phase*: each counter mapping in Γ enters *CmapRemoveQueue* at most once; hence it is removed from *CmapRemoveQueue* and Γ at most once. Each transition match is also removed at most once from *Tmatches* and *SuccCmapIndex*. All other operations are constant-time. Hence, the time complexity of this phase corresponds to the size of Γ and *Tmatches*: $O(n^2c! + n^2m^2c!^2) = O(n^2m^2c!^2)$.

We obtain a total (approximate) upper bound of time complexity $O(n^2m^2c!^2)$.

B.2 Size of SAT Formulation in Algorithm Search

1. **Variables.** The number of variables is given by the maximum number of counter mappings in $\lceil \sqsubseteq \rceil$, which equals the number of mappings in Γ – $O(n^2c!)$.

2. **Formulae.** The number and size of formulae for particular constraints is as follows:

   (a) *Unambiguity* – given by the number of mappings. $O(n^2c!)$ formulae, each of size $O(c!)$; total size $O(n^2c!^2)$.

   (b) *Reflexivity* – these constraints can effectively be ignored, as they are entirely excessive and serve only an "illustratory" purpose (along with the $\texttt{var}(\gamma_{\text{ID}}{}^{(q)})$ variables). The number of formulae is $\Theta(n)$, size is $\Theta(1)$ per each; total size is $\Theta(n)$.

   (c) *Transitivity* – there are n^3 possible (q, r, s) tuples, and for each tuple, $O(c!)$ possible mappings γ_q^r and $O(c!)$ possible γ_r^s. The total number of formulae is $O(n^3c!^2)$ and size is $\Theta(1)$ per each; total size is $O(n^3c!^2)$.

   (d) *Simulation* – we have $O(n^2c!)$ possible γ_q^s, hence $O(n^2c!)$ formulae. In each formula, there are $O(m)$ possible τ_q in the conjunction; each τ_q has $O(m)$ simulating τ_s and, in the successor states (q', s'), there are $O(c!)$ possible mappings $\gamma_{q'}^{s'}$. Hence the size of each formula is $O(m^2c!)$, and total size of all formulae is $O(n^2m^2c!^2)$.

   The total size of formulae is $O(n^3c!^2 + n^2m^2c!^2)$; their total count is $O(n^3c!^2)$.

Appendix C

Computing Bisimulations

With slight modifications, Algorithm Pseudosim and Algorithm Search can be used to compute *consistent bisimulations* instead of consistent simulations. In this appendix, we describe in general terms these modifications. Our main aim is simplicity; hence the resulting algorithms are not particularly efficient.

Note on inverse counter mappings. As we mentioned before, in the algorithms we use a shortened notation to denote a mapping associated with a specific pair of states. Instead of writing (q, s, γ), we simply write γ_q^s. When taking the inverse of a mapping, e.g. $(\gamma_q^s)^{-1}$, we must also switch the weaker and stronger state, so that – in the full notation – (q, s, γ) becomes (s, q, γ^{-1}) instead of (q, s, γ^{-1}).

C.1 Algorithm Pseudosim

We now describe a modification of Algorithm Pseudosim, which computes certain strongly-symmetric superset of the maximal hyperbisimulation on the input CA. We call this superset the *pseudobisimulation* and denote it $\lceil \doteqdot \rceil$. The structure of this section follows that of Sec. 5.1, describing in each subsection the necessary modifications of the computations presented in the original subsection.

C.1.1 Initialization Phase

Initialization Phase, Part 1

- After we obtain the structural FA A', we compute the *bisimulation equivalence* on A' instead of the simulation preorder $\preccurlyeq$. Once again, we are free to use any suitable algorithm for computing bisimulation equivalences on FA.
- When computing the initial overapproximation $\ddot{\Gamma}$ in Alg. 3, we consider a mapping valid only if it is "applicable in both directions". In particular, line 5 changes to

  $$\mathit{subsumptionHolds} := (\boldsymbol{min}_c = \boldsymbol{min}_d \wedge \boldsymbol{max}_c = \boldsymbol{max}_d);$$

  and line 6 changes to

  $$\mathit{uncondAcceptImplied} := ((\mathit{fin}_q(c) = \mathbf{true}) \iff (\mathit{fin}_s(d) = \mathbf{true})).$$

  (And these variables shall be renamed to reflect the change in their semantics.)

  This modification applies equally to the proposed optimization of Alg. 3 by finding matchings in a bipartite graph.

Initialization Phase, Part 2

After initializing the *Tmatches* and and *SuccCmapIndex* structures in Alg. 5, we must ensure that the sets $\ddot{\Gamma}^s_q$ and $\ddot{\Gamma}^q_s$ (for each q, s) are "inverse" of each other: $\ddot{\Gamma}^s_q = \{\gamma^{-1} \mid \gamma \in \ddot{\Gamma}^q_s\}$. This means we have to remove mappings from either of these sets, such that their inverse is not in the other set. As in Alg. 5, we cannot directly remove the mappings from $\ddot{\Gamma}$, as we need to perform maintenance when doing so. We instead add them to *CmapRemoveQueue* and they will be eventually removed during the inductive phase.

Algorithm 8 illustrates this operation. First, it adds to the queue all mappings such that their inverse does not exist is $\ddot{\Gamma}$ at all. Then it adds to the queue all mappings such that their inverse is about to be removed from $\ddot{\Gamma}$ (meaning it is in the queue).

Algorithm 8: Removing non-symmetric initial mappings before inductive phase

Input: *CmapRemoveQueue*, CA A
Effect: All non-symmetric mappings added to queue *CmapRemoveQueue*

```
1 for q, s ∈ Q × Q do
2 |   for γ^s_q ∈ Γ̈^s_q do
3 |   |   if (γ^s_q)^{-1} ∉ Γ̈^q_s then
4 |   |   |   CmapRemoveQueue := CmapRemoveQueue ∪ {γ^s_q};
5 for γ^s_q ∈ CmapRemoveQueue do
6 |   γ^q_s := (γ^s_q)^{-1};    // mapping is inverted and states are switched
7 |   if γ^q_s ∈ Γ̈^q_s then
8 |   |   CmapRemoveQueue := CmapRemoveQueue ∪ {γ^q_s};
```

C.1.2 Inductive Phase

We must ensure that every mapping in Γ^s_q has its inverse in Γ^q_s (for all q, s). This can be achieved by quite a simple modification of Alg. 6 – when adding a mapping to the queue of mappings to be removed, we also add its inverse (if it exists). Thus on line 17 we replace the added mapping $\{\gamma^{s'}_{q'}\}$ by $\{\gamma^{s'}_{q'}\} \cup (\{(\gamma^{s'}_{q'})^{-1}\} \cap \Gamma^{q'}_{s'})$. Due to this, Invariant 3 no longer holds in its original form on line 4. It holds, however, in a slightly modified form which we omit for its triviality. The remainder of the inductive phase remains the same; in particular, all remaining invariants still hold as before.

End of modified Algorithm Pseudosim. The result is the pseudobisimulation, which is then used by the modified Algorithm Search to compute consistent simulations.

C.2 Algorithm Search

The modification of Algorithm Search is a simple one. We extend the formulation with *symmetry constraints* which ensure that if a mapping exists in the solution, then so does its inverse. For each $q, s \in Q$ and $\gamma_q^s \in \lceil\sqsubseteq\rceil(q, s)$, we add to Φ the following formula:

$$\texttt{var(}\gamma_q^s\texttt{)} \Rightarrow \texttt{var(}(\gamma_q^s)^{-1}\texttt{)}.$$

Again, we note that the stronger and weaker states are switched in the inverse mapping $(\gamma_q^s)^{-1}$ (see the note on inverse counter mapping at the beginning of this appendix).

Observe that with these symmetry constraints alone, we are able to find bisimulations within the *pseudosimulation* computed by the unmodified Algorithm Pseudosim (i.e., we do not necessarily need the pseudobisimulation). However, that approach would be less efficient; hence the above modification of Algorithm Pseudosim.

End of modified Algorithm Search. We can now use the obtained bisimulation(s) for reduction of the input CA, as described in Chapter 4.

Appendix D

Additional Figures from Experiments

Here, we present additional figures concerning the experimental evaluation presented in Chapter 6, which display the obtained results in more detail.

Experiment 1: Figure D.1 shows the dependence of *absolute* simulation reduction on the number of states in the CA. The blue points are the mean of all y-values at the specified x-value; the blue bars represent the standard deviation. Keep in mind the y-scale is logarithmic, hence the deviation bars appear asymmetric but are in fact symmetric (i.e., the mean is in the middle of the deviation bar). The red points are medians. Observe that a clear majority of CA has an absolute reduction of 1 state, and this tendency is present even in larger CA (e.g. over 100 states).

Figure D.2 shows the dependence of *relative* simulation reduction on the number of states in the CA. Values (e.g. means, medians) are displayed precisely as in Fig. D.1.

Experiments 2 and 3: Figure D.3 shows the running time of the simulation algorithm as a function of the number of states. Displayed are values for runs which finished within the allotted time of 900 seconds (the 32 CA from Experiment 2 and 54 CA from Experiment 3). We did not perform a measurement of running time in Experiment 1, as the running time there was mostly negligible due to the small size of automata.

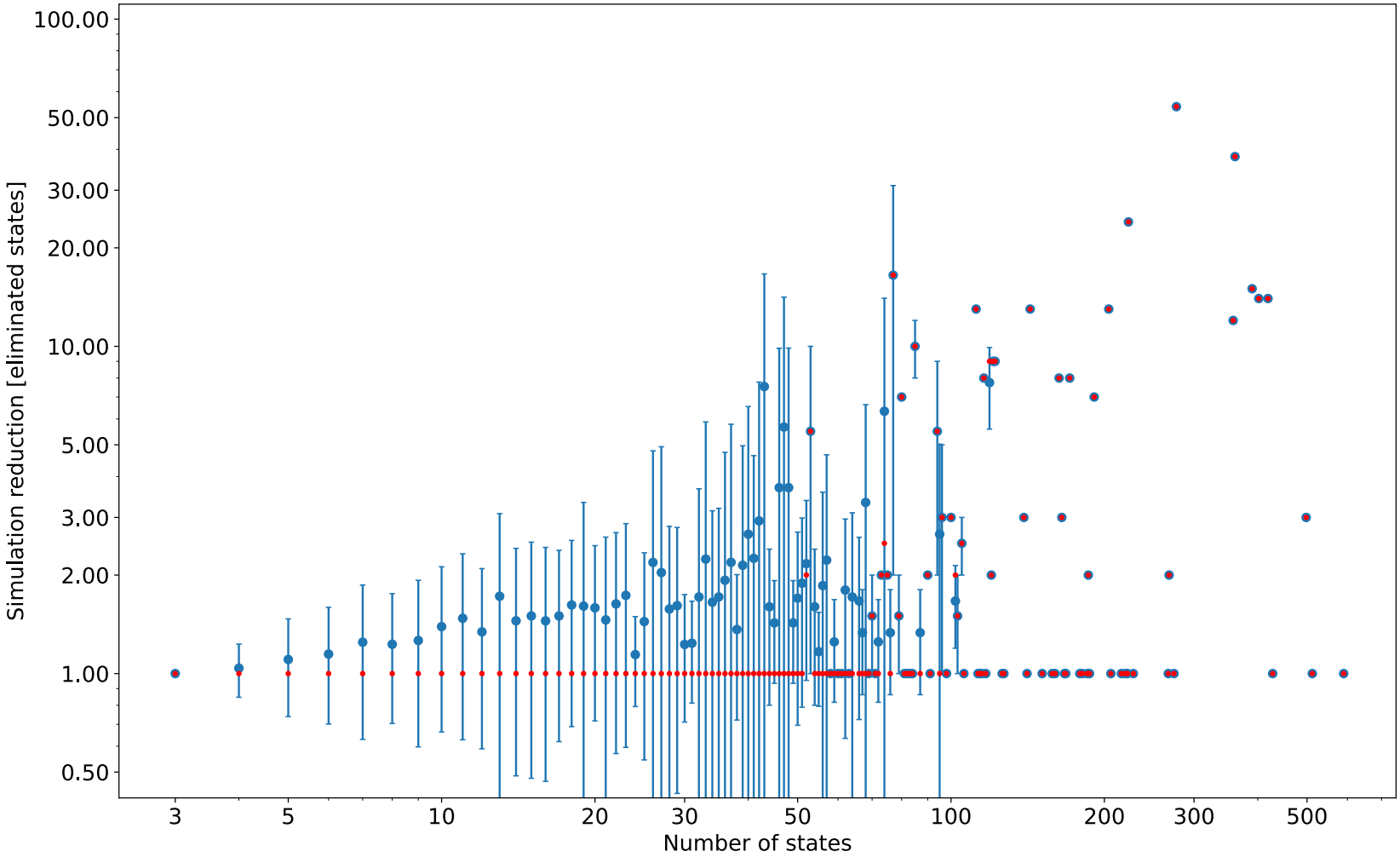


Figure D.1: (Experiment 1) Absolute simulation reduction as a function of the number of states.

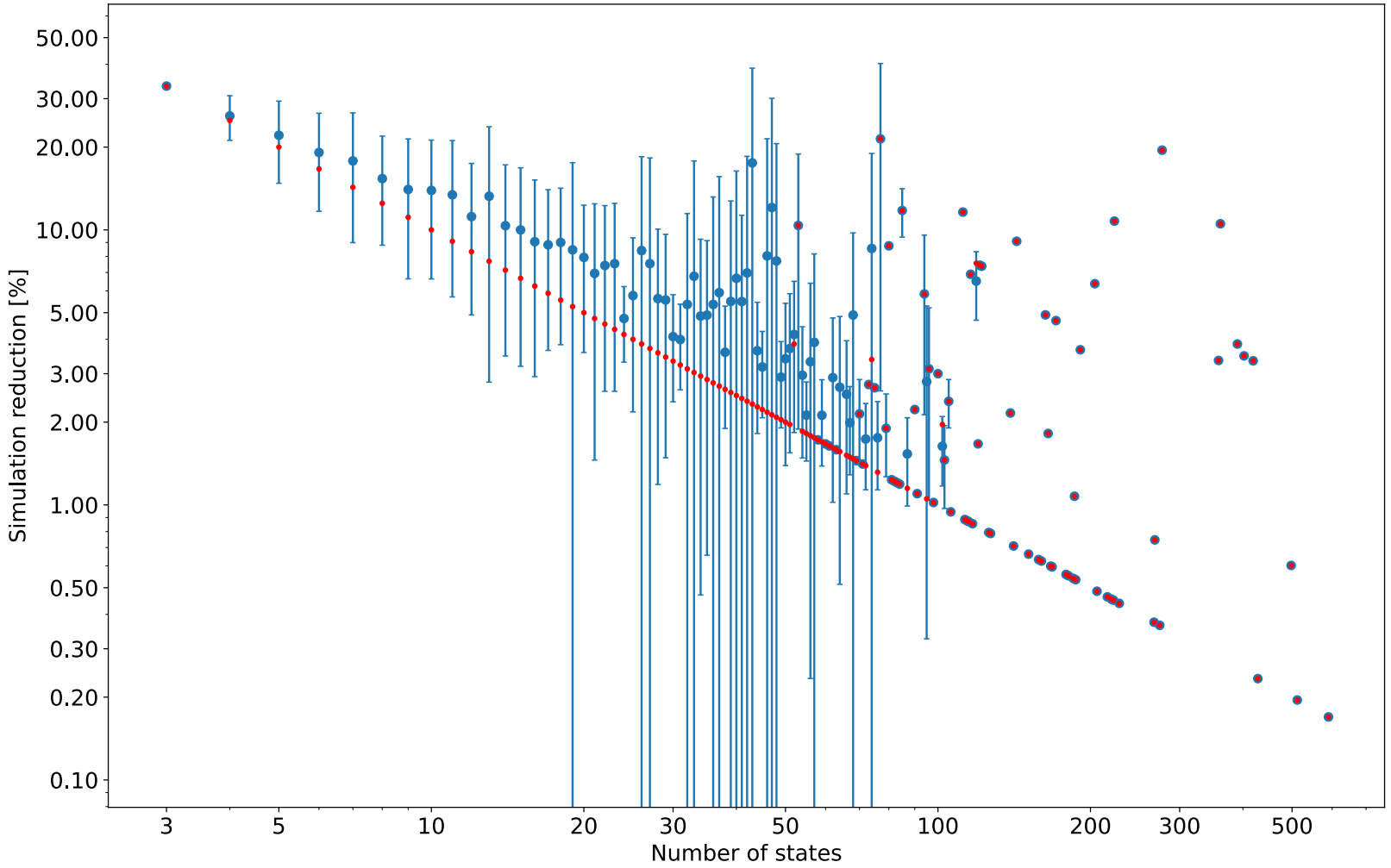


Figure D.2: (Experiment 1) Relative simulation reduction as a function of the number of states.

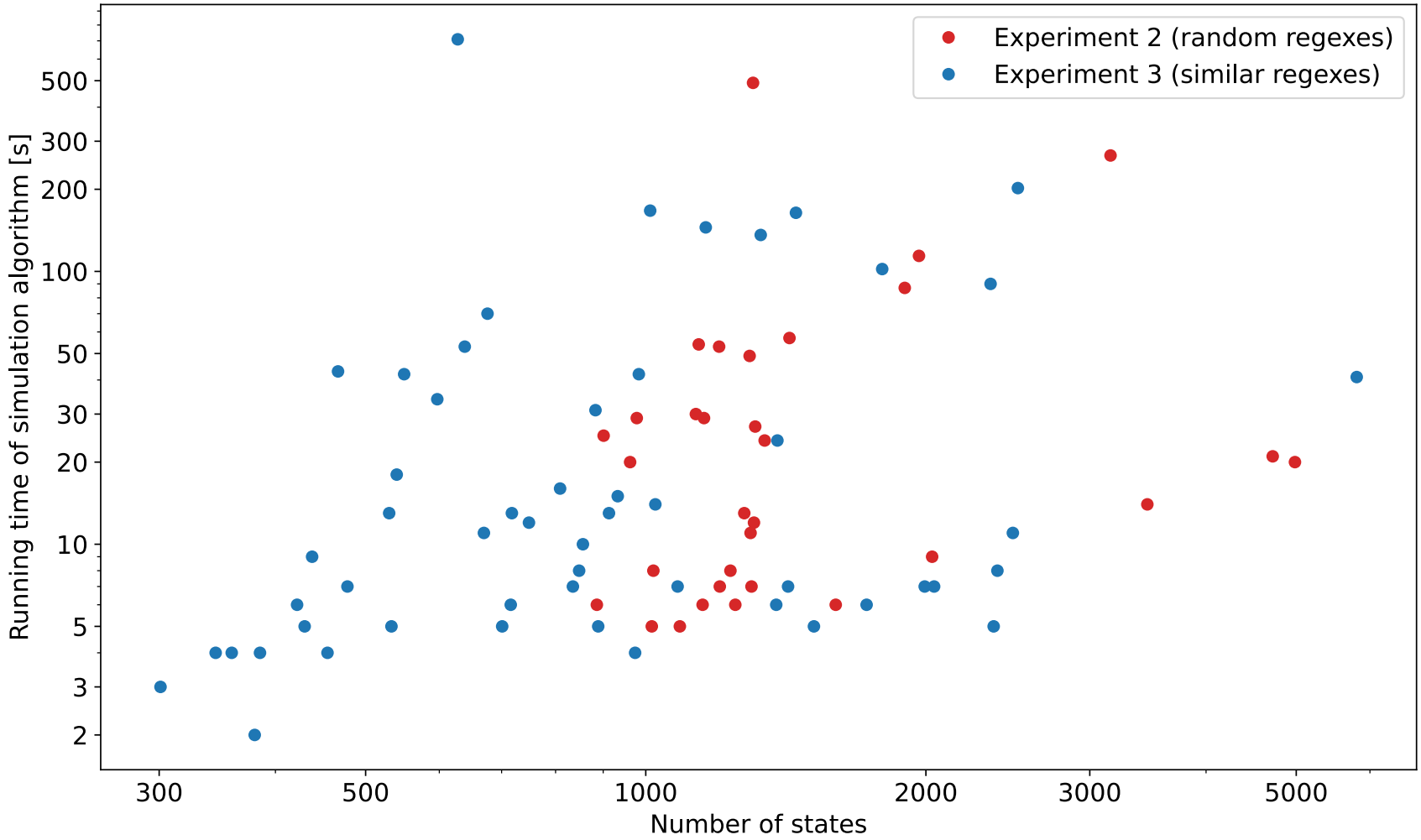


Figure D.3: (Experiments 2 and 3) Running time of the simulation algorithm as a function of the number of states.