BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TOOL FOR ABSTRACT REGULAR MODEL CHECKING

NÁSTROJ PRO ABSTRAKTNÍ REGULÁRNÍ MODEL CHECKING

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. MATĚJ CHALK**
AUTOR PRÁCE

SUPERVISOR — **Ing. MARTIN HRUŠKA**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání diplomové práce/21262/2017/xchalk00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Chalk Matěj, Bc.**
Obor: Matematické metody v informačních technologiích
Téma: **Nástoj pro abstraktní regulární model checking**
Tool for Abstract Regular Model Checking
Kategorie: Formální verifikace

Pokyny:

1. Nastudujte teorii konečných automatů a převodníků a jejich symbolických variant.
2. Nastudujte abstraktní regulární model checking.
3. Seznamte se s knihovnami pro práci se symbolickými automaty.
4. Vyberte vhodnou knihovnu a nad ní navrhněte nástroj pro abstraktní regulární model checking.
5. Implementujte navržený nástroj.
6. Otestujte na příkladech modelů uváděných v Bouajjani et al., např. Token Passing Protocol, případně i pekařův nebo Dijkstrův algoritmus a další.

Literatura:

- Christel Baier, and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.
- Loris D'Antoni, and Margus Veanes. The power of symbolic automata and transducers. *International Conference on Computer Aided Verification*. Springer, Cham, 2017
- A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular (Tree) Model Checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(2):167--191, Springer-Verlag, 2012.

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2, 3 a část bodu 4.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hruška Martin, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

Formal verification methods offer a large potential to provide automated software correctness checking (based on sound mathematical roots), which is of vital importance. One such technique is abstract regular model checking, which encodes sets of reachable configurations and one-step transitions between them using finite automata and transducers, respectively. Though this method addresses problems that are undecidable in general, it facilitates termination in many practical cases, while also significantly reducing the state space explosion problem. This is achieved by accelerating the computation of reachability sets using incrementally refinable abstractions, while eliminating spurious counterexamples caused by overapproximation using a counterexample-guided abstraction refinement technique. The aim of this thesis is to create a well designed tool for abstract regular model checking, which has so far only been implemented in prototypes. The new tool will model systems using symbolic automata and transducers instead of their (less concise) classic alternatives.

Abstrakt

Metody formální verifikace mohou poskytnout automatizované ověření korektnosti softwaru (stavěné na matematických základech), což je velmi důležité. Jednou z těchto metod je abstraktní regulární model checking, jenž používá konečné automaty a převodníky pro reprezentaci množiny dosažitelných konfigurací, respektive jednokrokového přechodu mezi těmito konfiguracemi. Přestože tato metoda řeší obecně nerozhodnutelné problémy, umožňuje terminaci v mnoha praktických případech a navíc výrazně zmírňuje problém stavové exploze. Tohoto dosahuje urychlením výpočtu dosažitelných stavů pomocí inkrementálního zjemňování abstrakcí, k odstranění neplatných protipříkladů vzniklých nadaproximací pak slouží technika zjemňování abstrakce založená na protipříkladech. Cílem této práce je vytvořit dobře navržený nástroj pro abstraktní regulární model checking, jenž byl dosud implementován pouze v prototypech. Nový nástroj bude systémy modelovat pomocí symbolických automatů a převodníků namísto jejich (méně stručných) klasických alternativ.

Keywords

formal verification, model checking, regular model checking, abstract regular model checking, automata, transducers, symbolic automata

Klíčová slova

formální verifikace, model checking, regulární model checking, abstraktní regulární model checking, automaty, převodníky, symbolické automaty

Reference

CHALK, Matěj. *Tool for Abstract Regular Model Checking.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Martin Hruška

Rozšířený abstrakt

Korektnost informačních a komunikačních systémů je velmi důležitá. Chyby v těchto systémech mohou vést k vysokým finančním ztrátám nebo ohrožení bezpečnosti. V červnu roku 1996 došlo k explozi nosné rakety Ariane-5 pouhých 36 vteřin po startu, protože řídící jednotka konvertovala 64-bitové číslo v plovoucí řádové čárce na 16-bitové celé číslo. V dubnu roku 2014 byla zveřejněna zranitelnost v kryptografické knihovně OpenSSL známá jako Heartbleed (odhalení citlivých dat kvůli chybějící kontrole délky pole), která byla v knihovně zavedena o dva roky dříve a vedla k útokům na 60 % webových serverů.

Pro produkci informačních a komunikačních systémů s minimálním množstvím chyb jsou zásadní efektivní techniky pro zajištění kvality. Mezi ně se řadí *peer-review* (posouzení zdrojového kódu vývojáři) a *testování* (spuštění softwaru s různými vstupy a kontrola výstupů). Obě techniky jsou velmi užitečné a používané, ale nemohou spolehlivě zajistit korektnost. Některé chyby (např. synchronizační, algoritmické) se těžko odhalují pomocí peer-review, zatímco vyčerpávající testování všech možných cest v programu je prakticky neproveditelné, a proto testování může odhalit pouze přítomnost chyb (absenci nikoliv).

Alternativou je užití technik *formální verifikace*. Formální verifikace je přirozeně stavěná na formálních matematických základech a narozdíl od ostatních technik má potenciál dokázat korektnost systémů vzhledem ke specifikaci požadovaných vlastností. Některé z těchto technik jsou navíc zcela automatizované, a tedy nevyžadují od uživatele mnoho interakce ani odborných znalostí. Lze rozlišit tři rozdílné přístupy k formální verifikaci, jmenovitě *model checking*, *statická analýza* a *theorem proving*. Statická analýza získává informace o chování systému z jeho zdrojového kódu bez nutnosti spouštění. Dokáže se vypořádat s rozsáhlými systémy, ale je často specializovaná pro konkrétní úkol. Theorem proving dokazuje teorémy odvozováním z axiomů, podobně jako v klasické výrokové logice. Je velmi obecný, ale vyžaduje značnou manuální práci.

Model checking je verifikační technika založená na modelech popisující systém a jeho chování matematicky přesným a jednoznačným způsobem. Pomocí těchto modelů, které mohou být automaticky generovány ze zdrojového kódu, se prochází všechny možné stavy systému vyčerpávajícím způsobem a kontroluje se, zda v některém z těchto stavů není porušena daná vlastnost. Při nalezení porušení vlastnosti se generuje protipříklad. Jelikož model checking verifikuje model systému, platnost jeho výsledků je závislá na kvalitě modelu.

Regulární model checking (RMC) používá jako své modely konečné automaty, které reprezentují (potenciálně nekonečnou) regulární množinu dosažitelných konfigurací ve zkoumaném systému. Přechody mezi konfiguracemi (tj. chování systému) jsou modelovány konečnými převodníky. Stejně jako ostatní techniky model checkingu, RMC se musí vypořádat s problémem stavové exploze (počet dosažitelných stavů roste exponenciálně). Navíc řeší verifikační úlohy, které jsou obecně nerozhodnutelné. Typicky se proto užívá nějaká metoda akcelerace, aby se zvýšila pravděpodobnost terminace.

Abstraktní regulární model checking (ARMC) dosahuje akcelerace použitím abstrakce, kterou často spojuje s technikou *zjemňování abstrakce založené na protipříkladech* (CEGAR). ARMC systematicky nadaproximuje množinu dosažitelných stavů, čímž zaručuje terminaci. Přináší tímto možnost objevení sporného protipříkladu (tj. chyba, která v systému skutečně není, ale byla nalezena kvůli nadaproximaci). Při detekci sporného protipříkladu umožňují efektivní techniky zjemnit abstrakci tak, aby se v další iteraci vyloučila možnost nalezení stejného protipříkladu.

Ačkoliv není garantováno, že vnější cyklus (iterativního zjemňování) někdy skončí, v mnoha praktických případech ARMC terminaci umožňuje. Navíc výrazně zmírňuje problém stavové exploze.

ARMC užívá dvou různých základních technik pro abstrakci konečných automatů. Obě jsou založené na sloučení některých stavů v automatu na základě nějaké relace ekvivalence. U první techniky se dva stavy považují se ekvivalentní, mají-li jejich jazyky neprázdný průnik se stejnými jazyky z množiny predikátových jazyků. U druhé techniky jsou stavy ekvivalentní, když si jsou rovné jejich jazyky o omezené délce slova. Zjemnění abstrakce se dosáhne v prvním případě přidáním predikátu a ve druhém navýšením délky. Pro obě techniky existuje řada různých variant a heuristik.

ARMC má velký potenciál poskytnout zcela automatizovanou verifikaci nekonečných a parametrizovaných systémů. Dosud byla však tato metoda implementována ve dvou prototypech v jazycích YAP Prolog a OCaml. Jelikož tyto prototypy již nejsou udržovány, nejsou vhodné pro experimentování a rozšiřování novými algoritmy.

Cílem této práce je návrh, implementace a testování plnohodnotného nástroje pro ARMC. Namísto konečných automatů a převodníků jsou pro reprezentaci systému použity symbolické automaty a převodníky (omezené na predikáty *in* a *not in* kvůli požadavku na uzavřenost inverze převodníku), které umožňují stručnější vyjadřování. Nový nástroj je implementován v jazyce C# a jako backend používá open source knihovnu *AutomataDotNet* od Microsoftu. Tato knihovna implementuje algoritmy nad symbolickými automaty a byla vybrána zejména kvůli její vyspělosti.

Nový nástroj je navržen objektově-orientovaným způsobem a může být použit jako knihovna nebo jako konzolová aplikace. Podporuje různé textové formáty pro načítání a tisknutí automatů/převodníků (Timbuk, FSA, FSM, pro tisk také DOT), a umožňuje zvolení konfigurace úpravou textového souboru v jednoduchém formátu (opatřeném komentáři). Nástroj je implementován tak, aby byl lépe udržovatelný. Na vhodných místech je užito funkcionálního programování a reflexe. Rozhraní je dokumentováno a zdrojový kód je okomentován.

Při testování se použily modely různých verifikačních úloh. Pomocí nového nástroje byla verifikována vlasnost vzájemného vyloučení procesů v kritické sekci u několika synchronizačních algoritmů (*pekařův*, *Dijkstrův*, *Burnsův* a *Szymańského algoritmus*), a také správné pořadí událostí u zásobníkového systému s rekurzivními procedurami (tvořící náhodný sloupcový graf) a komunikačního systému využívající fronty (*alternating bit protocol*). Navíc u chybné specifikace Szymańského algoritmu nalezl nástroj protipříklad. Experimentálně byla zjištěna velká závislost na použité konfiguraci.

Nový nástroj pro ARMC je navržen udržovatelným způsobem a má vhodné uživatelské rozhraní. V budoucnu je možné jej rozšířit o *abstraktní regulární stromový model checking*. Zdrojový kód je dostupný pod licencí MIT.

Tool for Abstract Regular Model Checking

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Martin Hruška. The supplementary information was provided by Prof. Ing. Tomáš Vojnar, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Matěj Chalk
May 21, 2018

Acknowledgements

I would like to thank my supervisor, Ing. Martin Hruška, for his helpful guidance, willing advice, and patience, as well as Prof. Ing. Tomáš Vojnar, Ph.D., for his useful input.

Contents

Chapter 1

Introduction

We rely heavily on the correct functioning of ICT systems (Information and Communication Technology). Errors in ICT systems may lead to big financial losses as well as to safety problems. In June 1996, the Ariane-5 missile notoriously crashed 36 seconds after launch due to a conversion of a 64-bit floating point number into a 16-bit integer value in its control software. In April 2014, a buffer over-read security bug (known as Heartbleed) in the OpenSSL cryptography library was publicly disclosed, two years after it was introduced into the software, with 60 % of web servers being attacked by hackers as a result.

Effective quality assurance techniques are critical for delivering low-defect ICT systems. Peer-review (review of source code by developers) and software testing (executing software with different inputs and checking results) are two useful and widely adopted quality assurance techniques, but both are unsound. Subtle errors such as concurrency and algorithm defects are hard to catch using peer review, while exhaustive testing of all execution paths is not practically feasible, meaning that testing can only uncover the presence of errors, not their absence.

An alternative is to use *formal verification* techniques. Formal verification is naturally based on formal, mathematical roots, and, unlike other techniques, is (at least potentially) capable of proving correctness of systems with regards to a given property specification. Some of these techniques also have the advantage of being fully automated, requiring very little user interaction or expertise. In the long term, formal methods offer a large potential to provide verification that is integrated early in the design process, is more effective and reduces verification time.

Three different approaches to formal verification and analysis may be distinguished, namely *model checking* [1] (which will be elaborated on shortly), *static analysis* [18] and *theorem proving* [22]. Static analysis collects information about system behaviour based on its source code without actually executing it, while theorem proving deductively proves theorems based on axioms and rules of inferrence in a similar way to classic propositional logic. The former can handle very large systems, but its analyses are often specialized for a specific task. The latter is very general, but requires a significant manual effort.

Model checking is a formal verification technique that is based on models describing the system behaviour in a mathematically precise and unambigious manner. These models, which may be automatically generated from the source code, are used to explore all possible system states in an exhaustive manner, and detect states for which a given property is

violated, in which case a counterexample is generated. Since model checking verifies a system model, any obtained result is only as good as the model [1].

Regular model checking (RMC) uses finite automata as its model, representing a (potentially infinite) regular set of reachable configurations of the system under consideration. Transitions between configurations (i.e. system behaviour) are modeled using finite transducers. Like all other model checking techniques, RMC has to contend with the state space explosion problem (the number of reachable states grows exponentially). Moreover, the verification task is undecidable in general. Various acceleration methods are typically used to increase the chances of termination.

Abstract regular model checking (ARMC) uses abstraction as its means of acceleration, often coupled with *counterexample-guided abstraction refinement* (CEGAR). ARMC systematically overapproximates the set of all reachable states in order to guarantee termination. This introduces the possibility of finding a spurious counterexample (i.e. an error which is not in fact present in the system, but has been encountered due to overapproximation). If a spurious counterexample is detected, effective techniques allow the abstraction to be refined in such a way as to exclude the possibility of encountering the same counterexample in the next iteration.

Although it is not guaranteed that the outer (refining) loop will terminate, ARMC facilitates termination in many practical cases, and also significantly reduces the state space explosion problem.

ARMC uses two different classes of techniques for abstracting finite automata, both of which are based on collapsing their states according to some equivalence relation. One bases its state equivalence relation on predicate languages, while the other bases it on finite-length languages [6].

ARMC offers a lot of potential for fully automated verification of infinite-state and parameterised systems. However, the only known implementations of this technique are in prototype tools, written in YAP Prolog and OCaml, respectively [6, 5]. Since these prototypes are no longer maintained, they are unsuitable for experimentation and the addition of new algorithms.

The aim of this thesis to design, implement and test a tool for ARMC. Instead of using finite automata and transducers to model a given system, a restricted version of symbolic automata and transducers will be used, which is more concise [8]. The new tool is written in C# and uses Microsoft's open-source library AutomataDotNet as its back-end [24]. This library implements algorithms over symbolic automata, and has been chosen for its maturity.

The text of this thesis first covers the necessary theoretical background. Chapter 2 describes the theory behind automata and transducers (both classic and symbolic), while Chapter 3 provides a description of abstract regular model checking. Following chapters then focus on the creation of the new ARMC tool. Chapter 4 discusses its design, Chapter 5 covers the implementation, and Chapter 6 describes how the tool was tested on specific verification tasks. The thesis ends with a conclusion in Chapter 7.

Chapter 2

Automata and Transducers

This chapter contains formal definitions and descriptions of relevant properties for finite automata and transducers (see Section 2.1), as well as their symbolic counterparts (see Section 2.2).

2.1 Finite Automata and Transducers

An *alphabet* is a non-empty set, whose elements are called *symbols* (or *letters*). A *word* (or *string*) over an alphabet Σ is defined according to the following rules:

(i) ε is a word over Σ, whose length is $|\varepsilon| = 0$,

(ii) if w is a word over Σ and $a \in \Sigma$, then wa is also a word over Σ and has a length of $|wa| = |w| + 1$.

2.1.1 Finite Automata

A *finite (state) automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

The transition relation $\xrightarrow[M]{} \subseteq Q \times \Sigma^* \times Q$ is the smallest relation satisfying:

(i) $\forall q \in Q\colon q \xrightarrow[M]{\varepsilon} q$,

(ii) if $q_2 \in \delta(q_1, a)$, then $q_1 \xrightarrow[M]{a} q_2$,

(iii) if $q_1 \xrightarrow[M]{w} q_2$ and $q_2 \xrightarrow[M]{a} q_3$, then $q_1 \xrightarrow[M]{wa} q_3$, where $w \in \Sigma^*, a \in \Sigma$.

The *(forward) state language* recognized by M from a state $q \in Q$ is defined as $L(M,q) = \{w \in \Sigma^* \mid \exists q_f \in F\colon q \xrightarrow[M]{w} q_f\}$, while the *backward state language* is defined as $\overleftarrow{L}(M,q) = \{w \in \Sigma^* \mid q_0 \xrightarrow[M]{w} q\}$. The overall *language* recognized by M is then defined as $L(M) = L(M,q_0)$. A set (or language) $L \subseteq \Sigma^*$ is *regular* iff it is recognized by some finite automaton M such that $L = L(M)$.

The *forward/backward state languages of words up to a certain length* are defined as $L^{\leq n}(M,q) = \{w \in L(M,q) \mid |w| \leq n\}$ and $\overleftarrow{L}^{\leq n}(M,q) = \{w \in \overleftarrow{L}(M,q) \mid |w| \leq n\}$, respectively.

The *forward/backward trace languages* of states are defined as $T(M,q) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^*\colon ww' \in L(M,q)\}$ and $\overleftarrow{T}(M,q) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^*\colon ww' \in \overleftarrow{L}(M,q)\}$, respectively. The *forward/backward trace languages of words up to a certain length* are then defined as $T^{\leq n}(M,q) = \{w \in T(M,q) \mid |w| \leq n\}$ and $\overleftarrow{T}^{\leq n}(M,q) = \{w \in \overleftarrow{T}(M,q) \mid |w| \leq n\}$, respectively.

If $\forall q \in Q\ \forall a \in \Sigma\colon |\delta(q,a)| \leq 1$, then M is a *deterministic finite automaton*, otherwise M is a *non-deterministic finite automaton*.

A state $q \in Q$ is *unreachable* iff $\overleftarrow{L}(M,q) = \emptyset$. States $q_1, q_2 \in Q$ are *nondistinguishable* iff $L(M,q_1) = L(M,q_2)$. M is a *minimal deterministic finite automaton* iff it is deterministic, contains no unreachable states and no two separate states are nondistinguishable.

As an example, Figure 2.1 shows a minimal deterministic finite automaton M such that $L(M) = \{a^m b^n c \mid m \in \{0,1\} \wedge n \geq 0\}$, $L(M,q_1) = \{b^n c \mid n \geq 0\}$, $\overleftarrow{L}(M,q_1) = \{a^m b^n \mid m \in \{0,1\} \wedge n \geq 1-m\}$, $L^{\leq 3}(M,q_1) = \{c, bc, bbc\}$, $\overleftarrow{L}^{\leq 2}(M,q_2) = \{c, ac, bc\}$, $T(M,q_1) = \{b^n c^p \mid n \geq 0 \wedge p \in \{0,1\}\}$, $\overleftarrow{T}(M,q_1) = \{a^m b^n \mid m \in \{0,1\} \wedge n \geq 0\}$, $T^{\leq 3}(M,q_1) = \{\varepsilon, b, c, bb, bc, bbb, bbc\}$ and $\overleftarrow{T}^{\leq 2}(M,q_2) = \{\varepsilon, a, b, c, ab, ac, bb, bc\}$.

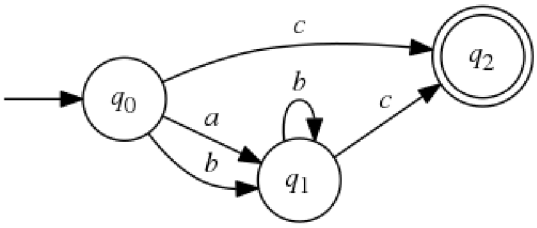


Figure 2.1: A finite automaton M.

For the purposes of abstracting automata later on, we define a *quotient automaton* $M_{/\sim} = (Q_{/\sim}, \Sigma, \delta_{/\sim}, [q_0]_{/\sim}, F_{/\sim})$ for an equivalence relation $\sim \subseteq Q \times Q$, where:

- $Q_{/\sim}$ and $F_{/\sim}$ are partitions of Q and F, respectively, with regards to $\sim$,
- $[q_0]_{/\sim}$ is the equivalence class of Q with regards to $\sim$ containing q_0,
- $\delta_{/\sim}$ is defined such that for all $[q_1]_{/\sim}, [q_2]_{/\sim} \in Q_{/\sim}$ and $a \in \Sigma$, it holds that $[q_2]_{/\sim} \in \delta_{/\sim}([q_1]_{/\sim}, a)$ iff $q_2 \in \delta(q_1, a)$ for some $q_1 \in [q_1]_{/\sim}, q_2 \in [q_2]_{/\sim}$.

Properties

Given a finite automaton $M = (Q, \Sigma, \delta, q_0, F)$, one can effectively construct a deterministic finite automaton $M_{det} = (Q', \Sigma', \delta', q_0', F')$ such that $L(M) = L(M_{det})$. Determinization may be performed using subset construction, i.e. $Q' = 2^Q$, $\Sigma' = \Sigma$, $\forall q' \in Q'\ \forall a \in \Sigma'\colon \delta'(q', a) = \bigcup_{q \in q'} \delta(q, a)$, $q_0' = \{q_0\}$ and $F' = \{q' \in Q' \mid q' \cap F \neq \emptyset\}$.

Given a finite automaton M, one can effectively construct a minimal deterministic finite automaton M_{min} such that $L(M) = L(M_{min})$, e.g. using Hopcroft's algorithm [11].

Given finite automata $M_1 = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_0^2, F_2)$, one can effectively construct a finite automaton $M_1 \times M_2 = (Q_\cap, \Sigma_\cap, \delta_\cap, q_0^\cap, F_\cap)$ such that $L(M_1 \times M_2) = L(M_1) \cap L(M_2)$. The intersection may be computed using product construction, i.e. $Q_\cap = Q_1 \times Q_2$, $\Sigma_\cap = \Sigma_1 \cap \Sigma_2$, $\forall (q_1, q_2) \in Q_\cap\ \forall a \in \Sigma_\cap\colon \delta_\cap((q_1, q_2), a) = \delta_1(q_1, a) \times \delta_2(q_2, a)$, $q_0^\cap = (q_0^1, q_0^2)$ and $F_\cap = F_1 \times F_2$.

Given finite automata M_1 and M_2, it is decidable to check if $L(M_1) = \emptyset$, and if $L(M_1) = L(M_2)$. Emptiness can be decided by checking if any final state is reachable, while language equivalence may be decided by minimizing both automata and checking that this results in the same automaton.

2.1.2 Finite Transducers

A *finite (state) transducer* is a 5-tuple $\tau = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states,
- Σ is a finite input/output alphabet,
- $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

The transition relation $\xrightarrow[\tau]{} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is the smallest relation satisfying:

(i) $\forall q \in Q\colon q \xrightarrow[\tau]{\varepsilon/\varepsilon} q$,

(ii) if $q_2 \in \delta(q_1, a_\varepsilon, b_\varepsilon)$, then $q_1 \xrightarrow[\tau]{a_\varepsilon/b_\varepsilon} q_2$,

(iii) if $q_1 \xrightarrow[\tau]{w/u} q_2$ and $q_2 \xrightarrow[\tau]{a_\varepsilon/b_\varepsilon} q_3$, then $q_1 \xrightarrow[\tau]{wa_\varepsilon/ub_\varepsilon} q_3$, where $w, u \in \Sigma^*, a_\varepsilon, b_\varepsilon \in (\Sigma \cup \{\varepsilon\})$.

The finite transducer τ defines the relation $\varrho_\tau = \{(w, u) \in \Sigma^* \times \Sigma^* \mid \exists q_f \in F\colon q_0 \xrightarrow[\tau]{w/u} q_f\}$. Given a finite automaton M, $\varrho_\tau(L(M))$ denotes the set $\{u \in \Sigma^* \mid \exists w \in L(M)\colon (w, u) \in \varrho_\tau\}$. For convenience, $\hat{\tau}(M)$ will be used to denote a minimal deterministic finite automaton M' such that $L(M') = \varrho_\tau(L(M))$.

The domain of τ is then defined as $\mathsf{dom}(\tau) = \{w \in \Sigma^* \mid \exists u \in \Sigma^*\colon (w, u) \in \varrho_\tau\}$, and the range of τ is defined as $\mathsf{ran}(\tau) = \{u \in \Sigma^* \mid \exists w \in \Sigma^*\colon (w, u) \in \varrho_\tau\}$.

As an example, Figure 2.2 shows a finite transducer τ such that $\mathsf{dom}(\tau) = \{HELLO!^n \mid n \geq 0\}$, $\mathsf{ran}(\tau) = \{Hello.\}$ and $\varrho_\tau = \mathsf{dom}(\tau) \times \mathsf{ran}(\tau)$.

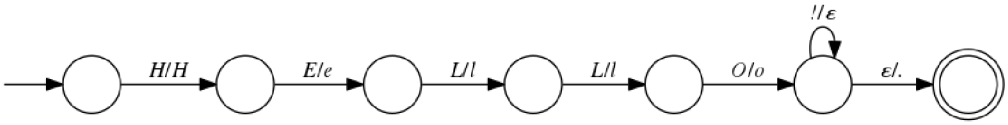


Figure 2.2: A finite transducer τ.

In general, a relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is a *regular relation* iff there exists a finite transducer τ such that $\varrho = \varrho_\tau$. A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is a *regularity preserving relation* iff $\varrho(L)$ is regular for every regular set $L \subseteq \Sigma^*$.

Properties

Given a finite transducer τ, one can effectively construct a finite transducer defining the inverse relation $\varrho_\tau^{-1} = \{(v, u) \mid (u, v) \in \varrho_\tau\}$. The construction consists of simply swapping the input and output symbols of all labels.

Given two finite transducers τ_1 and τ_2, one can construct a finite transducer $\tau_2(\tau_1)$ such that $\varrho_{\tau_2(\tau_1)} = \varrho_{\tau_2} \circ \varrho_{\tau_1} = \{(u, w) \mid \exists v\colon (u, v) \in \varrho_{\tau_1} \wedge (v, w) \in \varrho_{\tau_2}\}$. A product construction is used, with the transitions constructed such that $(q_2^1, q_2^2) \in \Delta((q_1^1, q_1^2), a, b) \iff \exists c\colon q_2^1 \in \Delta_1(q_1^1, a, c) \wedge q_2^2 \in \Delta_2(q_1^2, c, b)$.

2.2 Symbolic Automata and Transducers

2.2.1 Symbolic Automata

Formally, an *(effective) Boolean algebra* is an 8-tuple $\mathcal{A} = (\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$, where:

- $\mathcal{D}$ is a (potentially infinite) set of domain elements,
- Ψ is a set of predicates closed under the Boolean operators $\vee$, $\wedge$ and $\neg$, while also $\bot, \top \in \Psi$,
- $[\![\_]\!] : \Psi \to 2^{\mathcal{D}}$ is a denotation function st.
  - (i) $[\![\bot]\!] = \emptyset$,
  - (ii) $[\![\top]\!] = \mathcal{D}$,
  - (iii) $\forall \varphi, \psi \in \Psi\colon [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!], [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!], [\![\neg\varphi]\!] = \mathcal{D} \setminus [\![\varphi]\!]$.

It is required that checking satisfiability of any $\varphi \in \Psi$ (i.e. whether $[\![\varphi]\!] \neq \emptyset$ holds) is decidable.

Intuitively, a Boolean algebra is an interface for what kinds of predicates may appear in transition labels for an automaton. $\mathcal{D}$ corresponds to the alphabet, $\bot$ is a predicate that is unsatisfiable, $\top$ is a predicate that holds for every domain element (i.e. symbol in alphabet), and $[\![\varphi]\!]$ contains precisely those symbols that satisfy the predicate φ.

A *symbolic (finite) automaton* is a 5-tuple $M = (Q, \mathcal{A}, \Delta, q_0, F)$, where:

- Q is a finite set of states,
- $\mathcal{A} = (\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ is a Boolean algebra,
- $\Delta\colon Q \times \Psi \to 2^Q$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

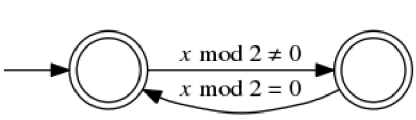
The transition relation $\xrightarrow[M]{} \subseteq Q \times \mathcal{D}^* \times Q$ is defined in the same way as for finite automata (see 2.1.1), with the exception of Σ being replaced by $\mathcal{D}$ and (ii) being changed to:

(ii) if $q_2 \in \Delta(q_1, \varphi)$ and $a \in [\![\varphi]\!]$, then $q_1 \xrightarrow[M]{a} q_2$.

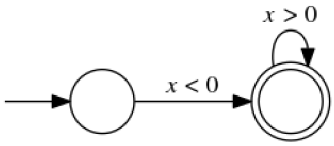
$L(M)$, $L(M,q)$, $\overleftarrow{L}(M,q)$, $L^{\leq n}(M,q)$, $\overleftarrow{L}^{\leq n}(M,q)$, $T(M,q)$, $\overleftarrow{T}(M,q)$, $T^{\leq n}(M,q)$ and $\overleftarrow{T}^{\leq n}(M,q)$ are then defined in the same way as for finite automata.

If $\forall q_1 \in \Delta(q,\varphi_1)\ \forall q_2 \in \Delta(q,\varphi_2)\colon q_1 \neq q_2 \implies [\![\varphi_1 \wedge \varphi_2]\!] = \emptyset$, then M is *deterministic*, otherwise M is *non-deterministic*.

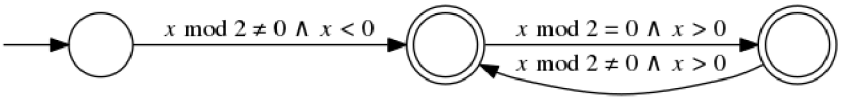
Figure 2.3 shows examples of symbolic automata, M_1 and M_2, which accept sequences of integers such that $L(M_1) = \{x_1, \dots x_n \mid n \geq 0 \wedge \forall 1 \leq i \leq n\colon i \bmod 2 = x_i \bmod 2\}$ and $L(M_2) = \{x_1, \dots x_n \mid n \geq 1 \wedge x_1 < 0 \wedge \forall 1 < i \leq n\colon x_i > 0\}$, as well a symbolic automaton $M_1 \times M_2$ accepting their intersection. For example, it holds that $(1,2,3,4,5) \in L(M_1) \setminus L(M_2)$.



(a) Symbolic automaton M_1.



(b) Symbolic automaton M_2.



(c) Symbolic automaton $M_1 \times M_2$ such that $L(M_1 \times M_2) = L(M_1) \cap L(M_2)$.

Figure 2.3: Examples of symbolic automata.

Properties

Given a symbolic automaton M, one can effectively compute a deterministic symbolic automaton M_{det} such that $L(M) = L(M_{det})$. Determiniziation works similarly to the subset construction for finite automata, but also requires combining predicates from different transitions by generating minterms.

Given a symbolic automaton $M = (Q, \mathcal{A}, \Delta, q_0, F)$, one can effectively construct a symbolic automaton $\overline{M}$ such that $L(\overline{M}) = \mathcal{D}^*_{\mathcal{A}} \setminus L(M)$. One must first add a non-final state $q_\bot$ such that

$\Delta(q_\bot, \top) = \{q_\bot\}$ and $\forall q \in Q$: $\Delta(q, \neg dom(q)) = \{q_\bot\}$ where $dom(q) = \bigvee\{\varphi \mid \exists q'\colon q' \in \Delta(q, \varphi)\}$. Swapping all final and non-final states will then result in the complement automaton.

Given two symbolic automata M_1 and M_2, one can effectively construct a symbolic automaton $M_1 \times M_2$ such that $L(M_1 \times M_2) = L(M_1) \cap L(M_2)$. The intersection may be computed using a version of the classic product construction in which transitions are combined using conjuction.

Given symbolic automata M_1 and M_2, it is decidable to check if $L(M_1) = \emptyset$, and if $L(M_1) = L(M_2)$. Emptiness can be decided by checking if a path from the initial state to some final state exists when ignoring unsatisfiable transitions. Language equivalence may then be reduced to checking emptiness of both set differences (note that $L(M_1) \setminus L(M_2) = L(M_1) \cap L(\overline{M_2})$).

2.2.2 Symbolic Transducers

Given a Boolean algebra $\mathcal{A} = (\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$, a set of *function terms* is denoted by Λ and a term $f \in \Lambda$ denotes an anonymous function $[\![f]\!]$ (which transforms an input symbol) over $\mathcal{D}$. The following holds for function terms:

- if $f, g \in \Lambda$, then $g(f) \in \Lambda$ and $\forall a \in \mathcal{D}$: $[\![g(f)]\!](a) = [\![g]\!]([\![f]\!](a))$,
- if $\varphi \in \Psi$ and $f \in \Lambda$, then $\varphi(f) \in \Psi$ and $\forall a \in \mathcal{D}$: $a \in [\![\varphi(f)]\!] \iff [\![f]\!](a) \in [\![\varphi]\!]$.

An (effective) Boolean algebra extended with function terms is called an *(effective) label algebra*.

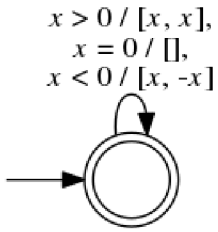
A *symbolic (finite) transducer* is a 5-tuple $\tau = (Q, \mathcal{A}, \Delta, q_0, F)$, where:

- Q is a finite set of states,
- $\mathcal{A}$ is a label algebra,
- $\Delta\colon Q \times \Psi \times \Lambda^* \to 2^Q$ is a transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

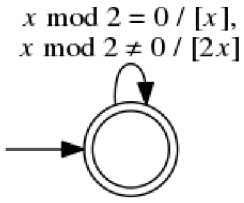
The transition relation $\underset{\tau}{\to} \subseteq Q \times \mathcal{D}^* \times \mathcal{D}^* \times Q$ is defined in the same way as for finite transducers (see 2.1.1), with the exception of Σ^* and $\Sigma \cup \{\varepsilon\}$ being replaced with $\mathcal{D}^*$ and $\mathcal{D}$, respectively, as well as (ii) being changed to:

(ii) if $q_2 \in \Delta(q_1, \varphi, f_1 \dots f_n)$ and $a \in [\![\varphi]\!]$, then $q_1 \xrightarrow[\tau]{a/[\![f_1]\!](a)\dots[\![f_n]\!](a)} q_2$.

Intuitively, a transition reads an input symbol that satisfies the predicate (or guard) and produces a sequence of output symbols by applying each function term to the input symbol. Figure 2.4 gives two examples of symbolic transducers, τ_1 and τ_2, which operate over the domain of integers. τ_1 deletes every 0 from a sequence, while copying every non-zero integer followed by its absolute value. τ_2 doubles every odd number, while leaving even numbers



(a) Symbolic transducer τ_1.



(b) Symbolic transducer τ_2.

Figure 2.4: Examples of symbolic tranducers.

unchanged. Given the input sequence $(1,0,-2,5,0)$, τ_1 produces $(1,1,-2,2,5,5)$, whereas τ_2 produces $(2,0,-2,10,0)$.

ϱ_τ, $\mathsf{dom}(\tau)$ and $\mathsf{ran}(\tau)$ are defined in the same way as for finite transducers, with the exception of Σ^* being replaced by $\mathcal{D}^*$.

Properties

Given two symbolic transducers τ_1 and τ_2, one can construct a symbolic transducer $\tau_2(\tau_1)$ such that $\varrho_{\tau_2(\tau_1)} = \varrho_{\tau_2} \circ \varrho_{\tau_1}$ (see Theorem 6 in [8]).

Given a symbolic transducer τ, one can compute a symbolic automaton DOM_τ such that $L(DOM_\tau) = \mathsf{dom}(\tau)$. However, the range of a symbolic transducer is in general not regular (see Theorem 5 in [8]).

A consequence of this is that, in general, it is not possible to construct a transducer for the inverse relation ϱ_τ^{-1}, as its domain (equal to $\mathsf{ran}(\tau)$) could not be regular. Since ARMC requires transducer inversion, we will define a restricted version of symbolic transducers (and automata), which is closed under inversion.

2.2.3 Simple Symbolic Automata

Let Σ be an alphabet. Then let the Boolean algebra $\mathcal{S}_\Sigma = (\mathcal{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ be defined as follows:

- $\mathcal{D} = \Sigma$,
- $\Psi = (\{\in, \notin\} \times 2^\Sigma) \cup \{\varepsilon\}$,
- $\forall (\in, A) \in \Psi$: $[\![(\in, A)]\!] = A$, $\forall (\notin, B) \in \Psi$: $[\![(\notin, B)]\!] = \Sigma \setminus B$ and $[\![\varepsilon]\!] = \varepsilon$.

For example, if $\Sigma = \{a, b, c\}$ then $\varphi_1 = (\in, \{a\})$, $\varphi_2 = (\notin, \{b, c\})$, $\varphi_3 = (\notin, \{a\})$, $\varphi_4 = (\in, \Sigma)$ and $\varphi_5 = (\notin, \emptyset)$ are all predicates in Ψ, with $[\![\varphi_1]\!] = [\![\varphi_2]\!] = \{a\}$, $[\![\varphi_3]\!] = \{b, c\}$ and $[\![\varphi_4]\!] = [\![\varphi_5]\!] = \Sigma$.

Let $M = (Q, \mathcal{S}_\Sigma, \Delta, q_0, F)$ be a symbolic (finite) automaton with $\mathcal{S}_\Sigma$ being the Boolean algebra defined above. We call M a *simple symbolic automaton*.

Figure 2.5 shows an example of a simple symbolic automaton over $\Sigma = \{a, \ldots z, A, \ldots Z, 0, \ldots 9, \_\}$, which recognizes identifiers in C-like programming languages.

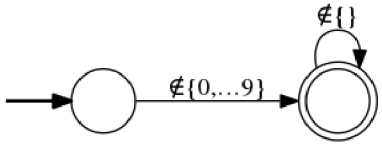


Figure 2.5: Example of a simple symbolic automaton.

Properties

Since a simple symbolic automaton is a special case of a symbolic automaton, it is clear that all problems that are decidable for symbolic automata are also decidable for simple symbolic automata.

Any finite automaton $M_{FA} = (Q, \Sigma, \delta, q_0, F)$ may be converted to a simple symbolic automaton $M_{SSA} = (Q, \mathcal{S}_\Sigma, \Delta, q_0, F)$ such that $L(M_{FA}) = L(M_{SSA})$ in the following way: $\forall q_1, q_2 \in Q (q_2 \in \Delta(q_1, \varepsilon) \iff q_2 \in \delta(q_1, \varepsilon)) \wedge (\forall a \in \Sigma\colon\ q_2 \in \Delta(q_1, (\in, \{a\})) \iff q_2 \in \delta(q_1, a))$.

Conversely, any simple symbolic automaton $M_{SSA} = (Q, \mathcal{S}_\Sigma, \Delta, q_0, F)$ may be converted to a finite automaton $M_{FA} = (Q, \Sigma, \delta, q_0, F)$ such that $L(M_{SSA}) = L(M_{FA})$ in the following way: $\forall q_1, q_2 \in Q\ \forall a \in \Sigma\colon\ q_2 \in \delta(q_1, a) \iff q_2 \in \Delta(q_1, \varphi) \wedge a \in [\![\varphi]\!]$ for some $\varphi \in \Psi$.

2.2.4 Simple Symbolic Transducers

A *simple symbolic transducer* is a 5-tuple $\tau = (Q, \mathcal{S}_\Sigma, \Delta, q_0, F)$, where:

- Q is a finite set of states,
- $\mathcal{S}_\Sigma$ is the same Boolean algebra over some alphabet Σ as for simple symbolic automata,
- $\Delta\colon\ Q \times \Psi \times (\Psi \cup \{\iota\}) \to 2^Q$ is a transition function $(\iota \notin \Psi)$,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

The transition relation $\underset{\tau}{\to} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined in the same way as for finite transducers (see 2.1.1), with the exception of (ii) being changed to:

(ii) for all $a_\varepsilon, b_\varepsilon \in (\Sigma \cup \{\varepsilon\})$ and $q_1, q_2 \in Q$, $q_1 \xrightarrow[\tau]{a_\varepsilon / b_\varepsilon} q_2$ if one of the following holds:

  (a) $q_2 \in \Delta(q_1, \varphi, \psi) \wedge \psi \neq \iota \wedge a_\varepsilon \in [\![\varphi]\!] \wedge b_\varepsilon \in [\![\psi]\!]$, or
  (b) $q_2 \in \Delta(q_1, \varphi, \iota) \wedge a_\varepsilon \in [\![\varphi]\!] \wedge a_\varepsilon = b_\varepsilon$.

Intuitively, a transition reads a symbol that satifies the input predicate, and either produces some symbol that satisfies the output predicate (non-deterministically), or, in the case of ι, copies the input symbol to the output (with ε meaning that nothing is read/writen). Unlike symbolic automata, transitions in simple symbolic automata only produce (at most) one output symbol instead of a sequence.

Figure 2.6 shows an example of a simple symbolic transducer over $\Sigma = \{a, \ldots z, A \ldots Z, 0, \ldots 9, \_\}$, which transforms an identifier written in underscore style to camel case (e.g. *model_checker* becomes *modelChecker*).

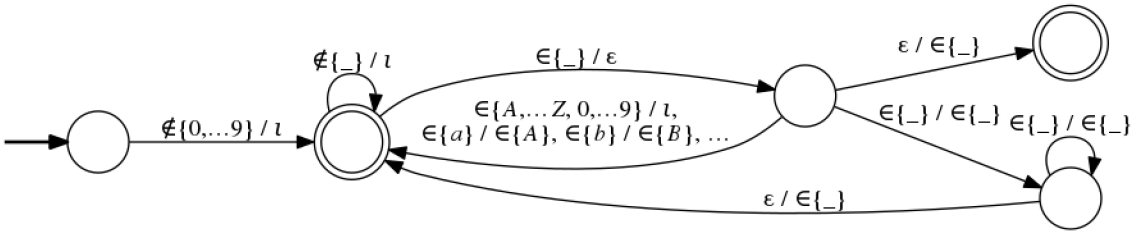


Figure 2.6: Example of a simple symbolic transducer.

Properties

Given two simple symbolic transducers τ_1 and τ_2, one can effectively construct a simple symbolic transducer defining the inverse relation $\varrho_{\tau_1}^{-1}$, as well as a simple symbolic transducer defining the composition $\varrho_{\tau_2} \circ \varrho_{\tau_1}$. Inversion is performed by swapping the input and output of all labels not containing ι. Composition is the same as for finite transducers, except that when composing labels φ_1/ψ_1 and φ_2/ψ_2 (from τ_1 and τ_2, respectively) the resulting label φ_1/ψ_2 is only created if $[\![\psi_1]\!] \cap [\![\varphi_2]\!] \neq \emptyset$.

Any finite transducer $\tau_{FT} = (Q, \Sigma, \delta, q_0, F)$ may be converted to a simple symbolic transducer $\tau_{SST} = (Q, \mathcal{S}_\Sigma, \Delta, q_0, F)$ such that $\varrho_{\tau_{FT}} = \varrho_{\tau_{SST}}$ in the following way: $\forall q_1, q_2 \in Q \ \forall a_\varepsilon, b_\varepsilon \in (\Sigma \cup \{\varepsilon\})$: $q_2 \in \Delta(q_1, f(a_\varepsilon), f(b_\varepsilon)) \iff q_2 \in \delta(q_1, a_\varepsilon, b_\varepsilon)$ where $f(\varepsilon) = \varepsilon$ and $\forall a \in Q : f(a) = (\in, \{a\})$.

Any simple symbolic transducer $\tau_{SST} = (Q, \mathcal{S}_\Sigma, \Delta, q_0, F)$ may be converted to a finite transducer $\tau_{FT} = (Q, \Sigma, \delta, q_0, F)$ such that $\varrho_{\tau_{SST}} = \varrho_{\tau_{FT}}$ in the following way: $\forall q_1, q_2 \in Q \ \forall \varphi, \psi \in \Psi \ \forall a_\varepsilon \in [\![\varphi]\!] \ \forall b_\varepsilon \in [\![\psi]\!]$: $(q_2 \in \delta(q_1, a_\varepsilon, b_\varepsilon) \iff q_2 \in \Delta(q_1, \varphi, \psi)) \wedge (q_2 \in \delta(q_1, a_\varepsilon, a_\varepsilon) \iff q_2 \in \Delta(q_1, \varphi, \iota)$.

Chapter 3

Abstract Regular Model Checking

This chapter describes the technique of abstract regular model checking. We first cover the basics of model checking (see Section 3.1) and regular model checking (see Section 3.2), before finally describing abstract regular model checking in detail (see Section 3.3).

3.1 Model Checking

Model checking is an automated technique that, given a model of a system and a formal property, systematically checks (by exploring the state space) whether this property holds for (every state in) that model [1].

The validity of the model is crucial for the whole verification process. Correct property specification is also important, and usually accomplished via the use a property specification language, with temporal logics (LTL, CTL, CTL*, etc.), which extend traditional propositional logic with operators referring to system behaviour over time, being a classic example.

The prerequisite inputs to model checking are an (accurate and unambiguous) model of the system being verified and a formal characterization of the property being checked. The model must be able to represent states or configurations of the system at any given time, as well as transitions between states which describe the system's behaviour.

In practice, constructing valid models can be rather difficult, because real-life systems tend to be complex and have informal specifications. Therefore, some simulations may be run before the actual model checking as a quick sanity check, which can help eliminate some simpler modeling errors. It is also possible to generate system models automatically from the source code.

The model checker must also be initialized by setting the various configuration options it may provide, as these can have a large effect on the efficiency of the exhaustive verification. The actual model checking algorithm is usually performed without user interaction.

There are three possible outcomes of model checking. The first is that the property is found to be valid, in which case the user may move on to check the next desired property. The second outcome is that the property is falsified (and diagnostic information is provided). This might be due to a modeling error (necessitating a correction of the invalid model and

a restart of the entire process), a property error (meaning that the incorrectly specified property must be modified and verified again), or a design error (which should be followed by improving the system design and model, before restarting the verification process). The third possible outcome of model checking is that the state space exceeds the physical limits of computer memory (this outcome is typical for infinite-state systems, which represent generally undecidable problems).

The last case is known as the state space explosion problem. The number of reachable states grows exponentially with regards to the source description of a finite-state system. For example, assuming an integer variable takes up 32 bits of memory (2^{32} possible values), if the system contains n such variables, the number of reachable states is $2^{n \cdot 32}$. Moreover, if the system is made up of m concurrent processes, the number of states reaches $2^{(n \cdot 32)^m}$. Given just 5 integer variables and 2 concurrent processes, the system can generate more states than the estimated amount of atoms in the universe.

There are several possible approaches for dealing with the state space explosion problem. One may store the state space efficiently (using hierarchical storage of states or binary decision diagrams), reduce the state space (using symmetries or partial order reduction), or use bounded model checking (exploring the state space up to some bound only, thus sacrificing soundness). Alternatively, one may also use abstraction (overapproximating the state space) and counterexample-guided abstraction refinement.

3.2 Regular Model Checking

Regular model checking is one of several approaches to model checking. The basic idea behind regular model checking is to encode system configurations as words over a finite alphabet, and represent potentially infinite sets of reachable configurations as regular languages, typically expressed using finite automata. Transitions between these sets of configurations are then encoded as one or more regularity preserving relations, typically expressed using finite transducers. Since several simple transducers may always be composed into a single more complex transducer, the rest of this section will assume a single transducer is used to encode a one-step transition relation of the system under consideration.

The RMC method takes three main parameters as its inputs:

- a finite transducer (or several transducers, which are combined into one) encoding the one-step transition relation ϱ of the system being verified,
- a finite automaton encoding the set I of all possible initial configurations of the system,
- a finite automaton encoding the set B of "bad" states, whose reachability in the given system is to be checked.

The goal of RMC is then to compute the set of all reachable configurations $\varrho^*(I)$, and check that no bad states are reachable, i.e. $\varrho^*(I) \cap B = \emptyset$. To obtain the reachability set $\varrho^*(I)$, one may repeatedly apply the transition relation ϱ to the set of reached states, and accumulate the union of all such sets. Formally, one may compute $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup \varrho(\varrho(\varrho(I))) \cup \ldots$. Alternatively, the reachability relation ϱ^* of the system may computed instead of $\varrho^*(I)$. In this case, one may repeatedly compose ϱ with the reachability relation computed thus far,

and take the union of all such relations. Or, in other words, $\varrho^* = \iota \cup \varrho \cup (\varrho \circ \varrho) \cup (\varrho \circ \varrho \circ \varrho) \cup \ldots$, where ι is the identity relation. In this case one checks that $\mathsf{ran}(\varrho^*) \cap B = \emptyset$, with $\mathsf{ran}(\varrho^*)$ denoting the range of the reachability relation.

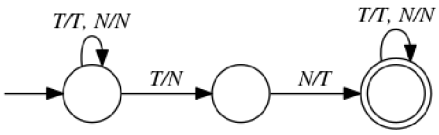
The computation of the reachability set $\varrho^*(I)$ may be terminated once a fixpoint has been reached, i.e. when the application of the transition relation ϱ results in the same set of reached states. However, when applied to parameterised and infinite-state systems, such a straightforward computation usually fails to terminate. This will be demonstrated shortly with a simple example.

3.2.1 Example: A Simple Token Passing Protocol

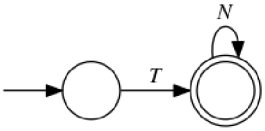
The RMC method will now be demonstrated using a simple example. Let us consider a parameterised network of processes (the number of processes is unknown, but finite) using synchronous communication to implement a *token passing protocol* to ensure mutual exclusion. A token is passed between processes arranged in a linear topology, and only a process that has the token is allowed entry into a critical section. Initially, only the leftmost process has the token. Every process awaits a token from its left neighbour, before passing the token to its right neighbour. It is required that no more than one process has the token at any given time.

The token passing protocol is modelled using the alphabet $\Sigma = \{T, N\}$. A configuration of the system then corresponds to a word $w = a_1 \ldots a_n \in \Sigma^*$ (n is the of number processes), where, for every i such that $1 \leq i \leq n$, a_i denotes whether the i-th process posseses a token ($a_i = T$), or not ($a_i = N$). The one-step transition relation ϱ is encoded by the finite transducer τ shown in Figure 3.1a, and represents the passing of a token possesed by some process to its right neighbour. The set of initial configurations I (only the leftmost process has a token) is represented by the finite automaton $Init$ shown in Figure 3.1b, while the set of bad states B (more than one token in the system) is encoded by the automaton Bad shown in Figure 3.1c.

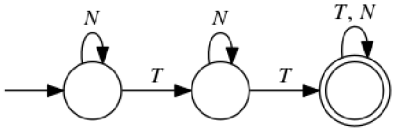
The goal is then to compute the reachability set $\varrho^*(I)$. Figure 3.2 illustrates an automaton accepting the set of reachable states. It is clear that this automaton has an empty intersection with Bad, meaning the property holds. However, when using the straightforward fixpoint computation of the union of infinitely many reachable states, the computation will



(a) The transducer τ such that $\varrho = \varrho_\tau$.



(b) The automaton $Init$ such that $I = L(Init)$.



(c) The automaton Bad such that $B = L(Bad)$.

Figure 3.1: Models for the simple token passing protocol.

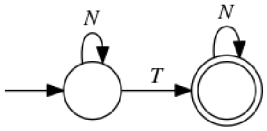


Figure 3.2: An automaton encoding the set of reachable configurations for the simple token passing protocol.

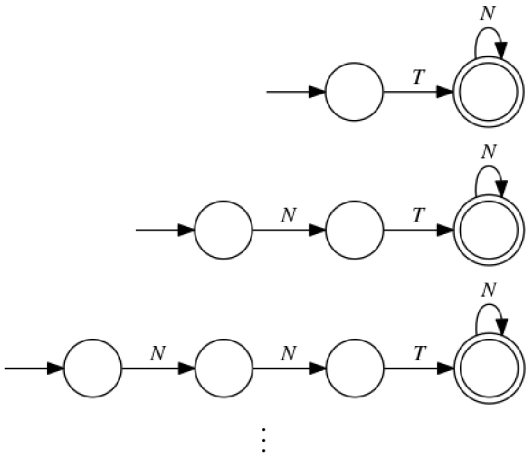


Figure 3.3: Automata constructed in the computation of the reachability set $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup \ldots$ for the simple token passing protocol.

not terminate. The first few sets are illustrated in Figure 3.3, and show that no fixpoint will ever be reached, since a new token position is found for every offset from the leftmost position (and there are infinitely many possible offsets).

The termination problem is typical for parameterised and infinite-state systems, even for this simple token passing protocol, and necessitates the need for some method of accelerating the computation.

3.3 Abstract Regular Model Checking

Abstract regular model checking is a formal verification technique which uses abstraction as a means of accelerating the computation of reachable states in the given system. The abstraction function overapproximates the sets of configurations in such a way as to guarantee termination. Since the reachability set is overapproximated (i.e. may contain states that are in fact unreachable, in addition to reachable states), encountered counterexamples may be spurious. Therefore, counterexample-guided abstraction refinement[1] is used to restart the computation after refining the abstraction so that the spurious counterexample will not be encountered again (Figure 3.4 illustrates the CEGAR loop). Though termination of the overall method is not guaranteed (abstraction refinement might go on forever), ARMC facilitates termination in many practical cases. Moreover, the state space explosion problem

[1]CEGAR may in fact be used for any model checking methods based on abstraction, e.g. predicate abstraction.

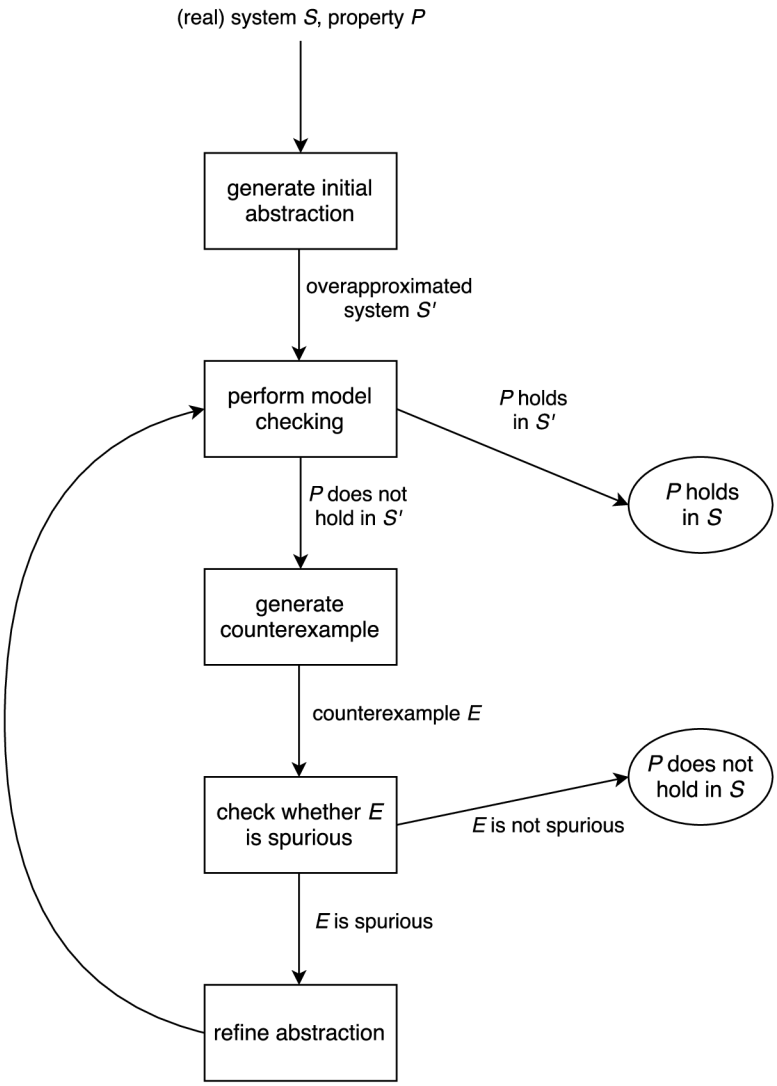


Figure 3.4: CEGAR loop.

is reduced significantly, by sidestepping one of its sources in RMC (calculating the exact reachability set, indepedently of the property being checked).

The abstraction is based on collapsing automata states according to some equivalence relation. ARMC considers two states to be equivalent when, either they have a non-empty intersection with the same predicate languages, or their state languages of words up to a certain length are equal, depeding on which abstraction technique is being used. Both of these techniques also provide efficient methods for refining the abstraction, and will described in detail later on.

3.3.1 The Method of Abstract Regular Model Checking

It will be assumed that τ refers to a finite transducer encoding the one-step transition in the system under consideration, while $Init$ and Bad refer to finite automata representing the initial set of reachable states and the set of "bad" states, respectively.

Let Σ be a finite alphabet. $\mathbb{M}_\Sigma$ will be used to denote the set of all finite automata over Σ, while $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ shall denote some abstract domain of automata. The automata abstraction function $\alpha\colon \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ is then defined such that $\forall M \in \mathbb{M}_\Sigma\colon L(M) \subseteq L(\alpha(M))$. In other words, α maps an automaton M to an automaton whose language is an overapproximation of the language of M.

The reachability relation ϱ_τ^* will be defined recursively in the following way. Let $\iota \subseteq \Sigma^* \times \Sigma^*$ be the identity relation. Then $\varrho_\tau^0 = \iota$, $\varrho_\tau^{i+1} = \varrho_\tau \circ \varrho_\tau^i$ and $\varrho_\tau^* = \bigcup_{i=0}^{\infty} \varrho_\tau^i$.

Next, let us define an abstract transition function τ_α such that $\forall M \in \mathbb{M}_\Sigma\colon \tau_\alpha(M) = \alpha(\hat{\tau}(M))$ (recall that $\hat{\tau}(M)$ denotes a minimal deterministic automaton accepting $\varrho_\tau(L(M))$). In order to iteratively compute the sequence of $\tau_\alpha^i(M)$ for all $i \geq 0$, two assumptions must be made to ensure that the computation will terminate. Let us suppose that $\iota \subseteq \varrho_\tau$ and that the range of our abstract transition function α (i.e. $\mathbb{A}_\Sigma$) is finite. This implies the existance of some $k \geq 0$ such that $\tau_\alpha^{k+1}(M) = \tau_\alpha^k(M)$. Given the definition of α, this means that in a finite number of steps, one may calculate a regular overapproximation of the reachability set, i.e. $\varrho_\tau^*(L(M)) \subseteq L(\tau_\alpha^k(M))$. Clearly, $L(\tau_\alpha^k(M)) \cap L(Bad) = \emptyset \implies \varrho_\tau^*(L(M)) \cap L(Bad) = \emptyset$.

Checking if an encountered counterexample is spurious is done in the following way. Let us assume for this abstract regular fixpoint computation that $L(Init) \cap L(Bad) = \emptyset$, otherwise the property being checked is broken in the initial configurations already. For each $i \geq 0$, let $M_i^\alpha = \alpha(M_i)$ and $M_{i+1} = \hat{\tau}(M_i^\alpha)$, where $M_0 = Init$. Building on the previous paragraph, there exists some $l > 0$ such that $\forall i\colon 0 \leq i < l : L(M_i) \cap L(Bad) = \emptyset$ and $L(M_l) \cap L(Bad) \neq \emptyset$, i.e. it is in the l-th iteration that the first possible property violation is encountered.

Then consider X_l to be an automaton such that $L(X_l) = L(M_l) \cap L(Bad)$, and for all i such that $0 \leq i < l$, X_i is a minimal deterministic automaton that accepts the language $\varrho_\tau^{-1}(L(X_{i+1})) \cap L(M_i^\alpha)$. The encountered counterexample is then spurious if there exists some k such that $0 \leq k < l$, for which it holds that $\forall i\colon k < i < l\colon L(X_i) \cap L(M_i) \neq \emptyset$ and $L(X_k) \cap L(M_k) = \emptyset$.[2] This

[2] At this point, either $k = 0$ or $L(X_{k-1}) = \emptyset$ (though $L(X_k) \neq \emptyset$).

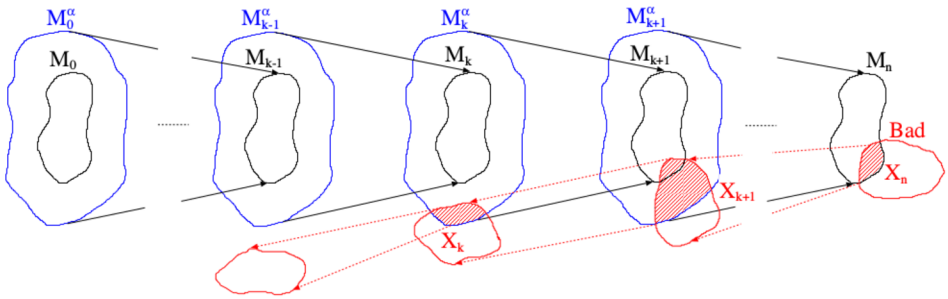


Figure 3.5: Detection of a spurious counterexample in a reachability computation.

```
ARMC(Init, Bad, τ)
begin
    if L(Init) ∩ L(Bad) ≠ ∅ then
        stop; property violated in initial configurations
    endif
    M_0 := Init
    loop # infinite loop, termination not guaranteed
        for i := 0 to ∞ do
            if L(M_i) ∩ L(Bad) ≠ ∅ then
                l := i
                let L(X_l) = L(M_i) ∩ L(Bad)
                break
            endif
            M_i^α := α(M_i)
            if i > 0 and L(M_i^α) = L(M_{i-1}^α) then
                stop; fixpoint reached, property holds
            endif
            M_{i+1} := τ̂(M_i^α)
        endfor
        for i := l − 1 downto 0 do
            let L(X_i) = ϱ_{τ^-1}(L(X_{i+1})) ∩ L(M_i^α)
            if L(X_i) ∩ L(M_i) = ∅ then
                k := i
                goto L
            endif
        endfor
        stop; property violated, generate counterexample (M_i, M_i^α, X_i where 0 ≤ i ≤ l)
      L:
        refine α (based on X_k or M_k)
    endloop
end ARMC
```

Listing 3.1: Pseudocode for the generic ARMC algorithm.

situation is depicted in Figure 3.5. On the other hand, if no such k exists (meaning that $L(X_0) \cap L(M_0) \neq \emptyset$), the property has been proven to hold.

An illustration of the ARMC method, at the current level of detail, is shown in Listing 3.1. Let us note that the algorithm may also be used in a backward computation, where we check that $(\varrho_\tau^{-1})^*(L(Bad)) \cap L(Init) = \emptyset$ instead (whereas the forward computation checks if $\varrho_\tau^*(L(Init)) \cap L(Bad) = \emptyset$).

In the case of a spurious counterexample, a refinement of α is needed. An automata abstraction function α' is a refinement of α iff $\forall M \in M_\Sigma$: $L(\alpha'(M)) \subseteq L(\alpha(M))$. The idea is that α' should be more precise than α. The spurious counterexample may be eliminated if the refinement is performed in such a way that for any automaton M, it holds that $L(M) \cap L(X_k) = \emptyset \implies L(\alpha'(M)) \cap L(X_k) = \emptyset$. This prevents M_k being abstracted to M_k^α again, thus avoiding a repetition of the same faulty sequence of M_i and M_i^α in the next abstract fixpoint computation. In fact the bad configurations will no longer be reachable, unless there was some reason for it other than overapproximating by subsets of $L(X_k)$. A weaker way of refinement may be used instead, where we allow that some subset of $L(X_k)$ may again be used for some overapproximation, but we at least ensure that $L(\alpha'(M_k)) \cap L(X_k) = \emptyset$ in

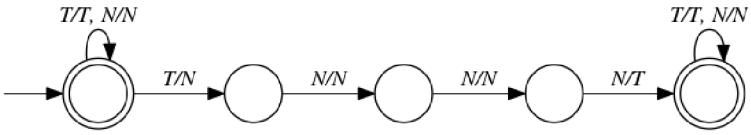
order to avoid the exact same faulty computation. In some cases, this may be more efficient than the stronger refinement.[3]

Automata abstraction functions are based on state equivalence schema, which define an equivalence relation on an automaton's states. The abstraction then consists of constructing a quotient automaton by collapsing equivalent states. Intuitively, two states are considered equivalent when their future or history is similar enough, and the difference may be abstracted away.

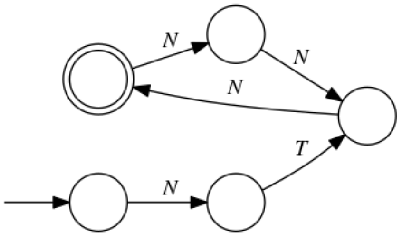
Formally, an automata state equivalence schema $\mathbb{E}$ assigns an automata state equivalence relation $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ to each finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ over Σ. The automata abstraction function $\alpha_{\mathbb{E}}$ based on $\mathbb{E}$ is then defined such that $\forall M \in \mathbb{M}_\Sigma$: $\alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$.

3.3.2 A Running Example

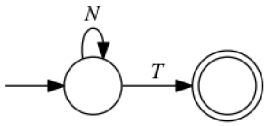
A slight modification of the simple token passing protocol described in 3.2.1 will be used as a running example to illustrate the different abstraction and refinement techniques in ARMC. In this modified version, each process passes the token to its third right neighbour, instead of its direct right neighbour (modeled by the transducer τ depicted in Figure 3.6a — note that $\iota \subseteq \varrho_\tau$). Initially, only the second leftmost process has the token, and the number of processes is a multiple of three (modeled by the automaton *Init* in Figure 3.6b). We wish to verify that it is not possible for the rightmost process to possess the token in any reachable configuration (modeled by *Bad* in Figure 3.6c).



(a) The transducer τ.



(b) The automaton *Init*.



(c) The automaton *Bad*.

Figure 3.6: Models for modified simple token passing protocol.

3.3.3 Abstraction Based on Predicate Languages

One way of abstracting automata is based on a set $\mathcal{P}$ of finite automata (called predicate automata). This approach leads to two automata state equivalence schemas – the schema $\mathbb{F}_\mathcal{P}$ based on forward state languages, and the schema $\mathbb{B}_\mathcal{P}$ based on backward state languages.

[3]The weaker refinement might lead to an earlier termination due to quickly jumping to the fixpoint, as well as less memory being used due to the sets of configurations being less structured. On the other hand, the coarser refinement may lead to more refinements down the road, and thus a slower computation.

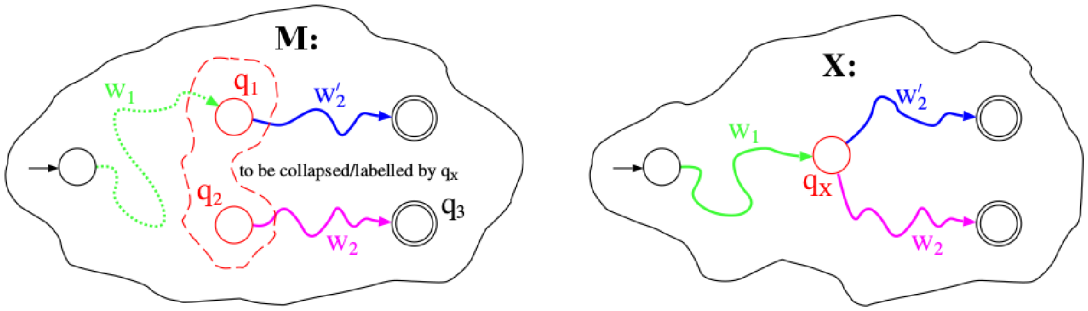


Figure 3.7: Proof sketch for predicate-based abstraction refinement.

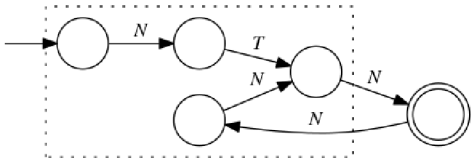
Two states are considered to be equivalent when their forward/backward state languages have a non-empty intersection with the same predicate automata in $\mathcal{P}$. The $\mathbb{F}_{\mathcal{P}}$ schema will be described in detail first, before a short description of how the $\mathbb{B}_{\mathcal{P}}$ schema differs.

Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_{\mathcal{P}}$ defines the equivalence $\sim_M^{\mathcal{P}}$ such that $\forall q_1, q_2 \in Q$: $q_1 \sim_M^{\mathcal{P}} q_2 \iff (\forall P \in \mathcal{P}$: $L(P) \cap L(M, q_1) \neq \emptyset \iff L(P) \cap L(M, q_2) \neq \emptyset)$. Given that $\mathcal{P}$ has a finite number of subsets, the range of $\alpha_{\mathbb{F}_{\mathcal{P}}}$ is finite.

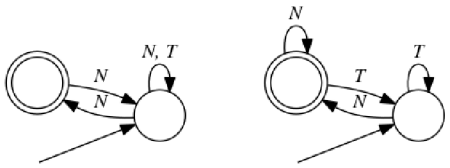
The $\mathbb{F}_{\mathcal{P}}$ schema may be refined by adding new predicates into the current set of predicates. Specifically, $\mathcal{P}$ may be extended with automata corresponding to the languages of each state in X_k from Figure 3.5. This technique prevents abstractions of M_k (or in fact any automata whose languages are disjoint with $L(X_k)$) from intersecting with X_k. The truth of this statement is shown in Theorem 1 in [6], which states that, for any two automata M and X, if $\forall q \in Q_X$: $\exists P \in \mathcal{P}$: $L(P) = L(X, q)$ and $L(M) \cap L(X) = \emptyset$, then $L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X) = \emptyset$ too. The theorem is proved by contradiction. The initial assumption is that there exists some $w \in L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X)$, and moreover $w = w_1 w_2$ is such a word that requires the minimum number of "jumps" between equivalent states (with regards to $\sim_M^{\mathcal{P}}$) to be accepted in M, with the last jump being from q_1 to q_2, after which w_2 is accepted. Let q_X be the state X is in after reading w_1 (note that $w \in L(X)$), then since $L(X, q_X) \in \mathcal{P}$, $w_2 \in L(M, q_2) \cap L(X, q_X)$ and $q_1 \sim_M^{\mathcal{P}} q_2$, there must exist some $w_2' \in L(M, q_1) \cap L(X, q_X)$ (because both q_1 and q_2 must have a non-empty intersection with the predicate $L(X, q_X)$). However, this means that an even smaller number of jumps is needed for $w_1 w_2' \in L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X)$ (since the last jump is avoided), which is a contradiction. Figure 3.7 illustrates this proof.

Let us apply this abstraction technique to our running example (see Figure 3.6), with the set $\mathcal{P}$ initially being made up of the languages of all the states of Bad. When $Init$ is first abstracted (illustrated in Figure 3.8a), all its states except the final one are considered equivalent (they have empty intersections with both states of Bad), resulting in the automaton depicted in Figure 3.8b. When the transducer τ is applied to $\alpha(Init)$, the resulting automaton (depicted in Figure 3.8c) has a non-empty intersection with the bad configurations (as shown in Figure 3.8d).

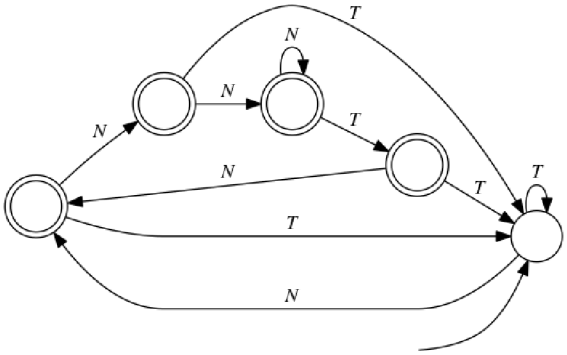
Since the counterexample turns out to be spurious, the abstraction is refined by adding all the state languages in X_0 (depicted in Figure 3.8e). With this more refined abstraction in place, only two states of $Init$ are now considered equivalent (shown in Figure 3.8f), and the result of the abstraction is depicted in Figure 3.8g. This is a fixpoint, and so the property has been verified.



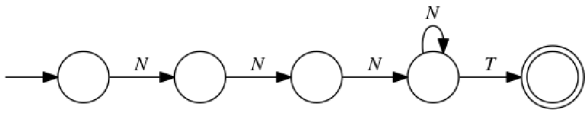
(a) Initial abstraction of $Init$.



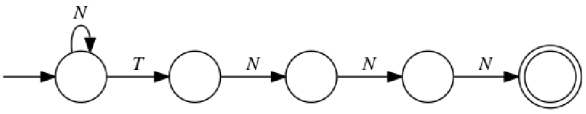
(b) $\alpha(Init)$ before (left) and after (right) determinization and minimization.



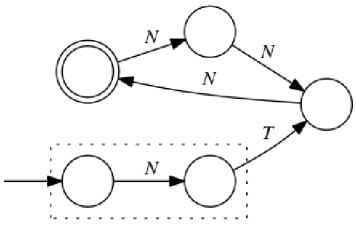
(c) $\hat{\tau}(\alpha(Init))$.



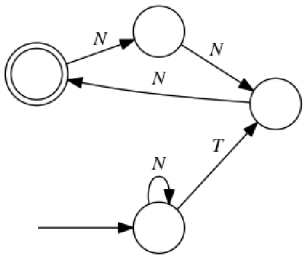
(d) X_1 such that $L(X_1) = L(\alpha(Init)) \cap L(Bad)$.



(e) X_0 such that $L(X_0) = \varrho_\tau^{-1}(X_1) \cap L(\alpha(Init))$.



(f) Refined abstraction of $Init$.



(g) Result of refined abstraction of $Init$.

Figure 3.8: Example of abstraction and refinement based on predicate languages.

The $\mathbb{B}_{\mathcal{P}}$ automata state equivalence schema is much like $\mathbb{F}_{\mathcal{P}}$, except that it uses backward state languages instead of forward ones. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, the equivalence $\overleftarrow{\sim}^{\mathcal{P}}_M$ is defined such that $\forall q_1, q_2 \in Q$: $q_1 \overleftarrow{\sim}^{\mathcal{P}}_M q_2 \iff (\forall P \in \mathcal{P}$: $L(P) \cap \overleftarrow{L}(M, q_1) \neq \emptyset \iff L(P) \cap \overleftarrow{L}(M, q_2) \neq \emptyset)$. Clearly, the range of $\alpha_{\mathbb{B}_{\mathcal{P}}}$ is finite for the same reason as for $\alpha_{\mathbb{F}_{\mathcal{P}}}$. The refinement by adding state languages of X_k has the same effect for $\mathbb{B}_{\mathcal{P}}$, as is shown in Theorem 2 in [6]. The proof is analogous to the proof for $\mathbb{F}_{\mathcal{P}}$, the difference being that, given the use of backward languages, the first jump and a prefix are considered instead of the last jump and a suffix.

We may sometimes speed up the computation by using a weaker refinement, as discussed in 3.3.1. In this case, we only consider the *important tail/head part* of X_k with regards to M_k, i.e. we only add automata corresponding to the state languages that have a non-empty intersection with some state language of M_k.

A further possible heuristic is to only consider one or two *key states* of the important tail/-head part of X_k, such that when the abstraction is refined by adding only their languages to $\mathcal{P}$, the thusly refined abstraction of M_k will no longer intersect with $L(X_k)$. This does not guarantee the exclusion of the same counterexample at all, presenting the danger of looping, but may lead to a faster computation in some cases.

For both $\mathbb{F}_{\mathcal{P}}$ and $\mathbb{B}_{\mathcal{P}}$, the initial set of predicates $\mathcal{P}$ may consist of all the state languages of the automata encoding the set of initial and/or bad configurations. Additionally, one may include automata which accept the domain and/or range of transducers in the system (of which the one-step transducer used in iteration is a union).

3.3.4 Abstraction Based on Finite-Length Languages

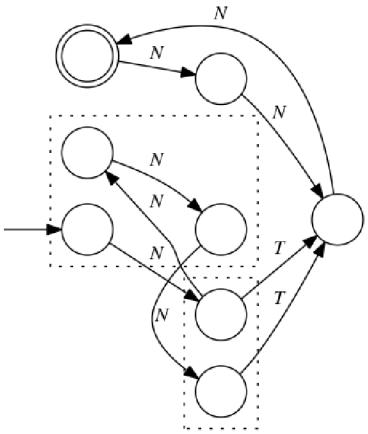
A different approach to abstraction and refinement in ARMC is based on languages of words up to certain length. Two states are considered equivalent if they represent the same language when given an upper limit n. Several different alternatives are possible. One may base the equivalence on forward state languages (the $\mathbb{F}^L_n$ automata state equivalence schema), backward state languages (the $\mathbb{B}^L_n$ schema), forward trace languages ($\mathbb{F}^T_n$), or backward trace languages ($\mathbb{B}^T_n$)[4]. Only $\mathbb{F}^L_n$ will be defined here, as the other three alternatives are analogous.

Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, the $\mathbb{F}^L_n$ schema defines the state equivalence $\sim^n_M$ such that $\forall q_1, q_2 \in Q$: $q_1 \sim^n_M q_2 \iff L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$. The range of $\alpha_{\mathbb{F}^L_n}$ is clearly finite.

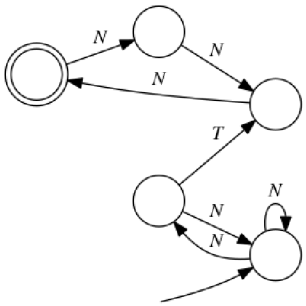
The abstraction may be refined by incrementing the bound n. This may lead to the weaker form of refinement described above, if n is increased by to be (at least) as large as the number of states in M_k minus one. This suffices because, in a minimal deterministic automaton, this means all the states will be distinguishable with regards to $\sim^n_M$, and M_k will therefore not be collapsed.

Given our running example (see Figure 3.6) and an initial value of n equal to 2, $Init$ will not be collapsed at all, but $\hat{\tau}(Init)$ will be collapsed (shown in Figure 3.9a), resulting in the automaton in Figure 3.9b, which is a fixpoint.

[4] It should be noted that the schemas based on forward/backward trace languages ($\mathbb{F}^T_n$ and $\mathbb{B}^T_n$), while reportedly useful in practice [6], do not guarantee the exclusion of a spurious counterexample.



(a) Abstraction of $\hat{\tau}(Init)$.



(b) $\alpha(\hat{\tau}(Init))$.

Figure 3.9: Example of abstraction and refinement based on finite-length languages.

When increasing n, one need not choose an increment of $|Q_{M_k}|-1$ (which may often be too large in practice, according to [6]), but may instead choose a fraction (e.g. one half), the number of states in X_k (or a fraction), or simply increment by 1. Since n will eventually reach the necessary value, an (eventual) exclusion of the same faulty computation is still guaranteed. Similarly, the initial value of n might be set to the number of states in the automaton encoding the initial configurations, or the bad configurations, a fraction, or 1.

Chapter 4

Tool Design

This chapter covers the design of the new abstract regular model checking tool. The tool may be used as a console application as well as a C# library. All classes are enclosed within an `ARMC` namespace, and errors are handled by throwing an instance of `ARMCException` with a descriptive message.

Section 4.1 justifies the use of simple symbolic automata and transducers instead of alternatives presented in Chapter 2. Section 4.2 then reports on the reasoning behind choosing the AutomataDotNet library as a back-end for algorithms over symbolic automata. Subsequently, the interface and class hierarchy of the new tool is described, with Section 4.3 focusing on automata and transducers, Section 4.4 detailing the configuration process and capabilities, Section 4.5 describing the polymorphic approach taken to handling different abstraction techniques, and Section 4.6 putting it all together in a description of the main interface for running ARMC.

4.1 The Case for (Simple) Symbolic Automata

Although ARMC is defined using finite automata and transducers, the new tool uses symbolic automata and transducers instead. Symbolic automata have recently become very popular, their main advantage being that they are highly expresive. They also retain many of the good properties of finite automata — efficient algorithms exist for their determinization and minimization, complement and intersection computation, as well as deciding emptiness and language-equivalence [8]. On top of that, they can also operate on infinite domains (e.g. the set of natural numbers).

As far as incorporating them into the ARMC method is concerned, there is one limitation which prevents symbolic automata, or specifically symbolic transducers, being used in their most general form. The problem is that given a transducer τ representing the relation ϱ_τ, ARMC requires that one should be able to compute the inverse relation ϱ_τ^{-1} (used when tracking back to determine whether a counterexample is spurious). However, as shown in 2.2.1, symbolic transducers are not closed under inversion.

Therefore, we will instead use simple symbolic automata and transducers (defined in 2.2.1 and 2.2.2, respectively). They use a restricted set of predicates, allowing queries of whether the input symbol is *in* (or *not in*) a finite set. These predicates are inspired by the FSA

library [20]. Though this version of symbolic automata does not offer greater expresiveness than classic automata, they are more concise as they allow multiple transitions to be merged into one. For example, given an alphabet $\Sigma = \{a, \ldots z\}$, if one wants to accept a consonant, a single transition will suffice with the label $\notin \{a, e, i, o, u\}$, instead of the 21 transitions one would have to specify using finite automata.

4.2 Choosing a Symbolic Automata Library

Four different libraries, all open source, were considered for providing implementations of algorithms for symbolic automata and transducers. Determinization, minimization, intersection computation, emptiness and language equivalence checking are among the main automata algorithms required by ARMC, as well as composition of transducers, and applying transducers to automata. ARMC also requires the construction of automata accepting forward/backward state/trace (finite-length) languages, and quotient automata. None of the considered libraries provide all of these algorithms, but as some of them are quite simple to implement, the focus was on the most difficult and intensive algorithms, e.g. minimization.

One of the considered libraries was *symboliclib*, which provides many algorithms operating over simple symbolic automata and transducers, with the explicit intent of being used for formal verification purposes [3, 4]. It uses state-of-the-art algorithms for language inclusion checking (which can be used to check language equivalence), specifically simulations and antichains. It is implemented in Python, and the code is written in a very straightforward and understandable way, meaning that it has a very quick learning curve. In the process of considering the suitability of this library, a small Python prototype of the ARMC method was implemented using symboliclib. This helped reveal several bugs in the library's implementation that had to be fixed in order to get the prototype up and running. In addition to the presence of bugs, there was also a concern over Python being a suitable language for an algorithm as computationally demanding as ARMC.

The other main candidate was the *AutomataDotNet* library, written by Margus Veanes for Microsoft using C# [24]. The library implements the most general version of symbolic automata and transducers, and optionally uses a Z3 solver. It is reportedly used by Microsoft in production, making it the most mature of the considered libraries. The code base is very large, and is composed of many interacting modules, which means that it has a very steep learning curve. On the other hand, it is very well designed, using generic data types to allow the user to provide their own Boolean algebra over which automata operate. Though the implementation of symbolic transducers is ill suited for simple symbolic transducers, both symbolic automata and symbolic transducer classes inherit from a parent automaton class, which in facts implements all the core automata algorithms. It is therefore possible to use this parent class in order to define simple symbolic automata and transducers without much inconvenience.

The *automata* library was shortly considered too [27, 26]. Written in the functional programming language Haskell, it claims to implement efficient algorithms over finite automata. Some groundwork has also been laid to accomodate finite tree automata, which could be useful if the ARMC tool were extended to include abstract regular tree model checking as

well. However, this library does not implement transducers or any symbolic versions, and it is also unclear whether it actually succeeded in implementing its algorithms efficiently.

Lastly, the *VATA* library was also among the candidates at one point [15]. VATA is implemented in C++, and provides highly optimized algorithms for finite automata, as well as for finite tree automata. Its efficiency with regards to speed is its main advantage, but it is also very complex and hard to use. Moreover, it does not implement transducers or symbolic automata.

In the end, the AutomataDotNet library was deemed most suitable. It was chosen mainly for its maturity, completeness, and good design with regards to implementing special cases of symbolic automata.

4.3 Automata and Transducers

The core of the AutomataDotNet library provides algorithms for symbolic automata and transducers, which are represented by the classes `SFA` and `ST`, respectively. Both of these classes contain an instance of an `Automaton` class, which defines the basic structure for an automaton (initial state, final states, transitions), as well as implementing core algorithms. These algorithms include determinization, minimization, equivalence checking and product construction among others. The `SFA` and `ST` classes provide many wrapper methods for these algorithms, as well associating the automaton with an SMT solver.

Since the new tool need by necessity be restricted to using simple symbolic automata/transducers (see 4.1), the `SFA` and `ST` classes are unsuitable. Both are unnecessarilly complex for our purposes and the definition of a general symbolic transducer is incompatible with the simpler version (e.g. labels having several yields). On the other hand, the core `Automaton` class contains all the implementations for the algorithms provided[1], and has a simpler interface. It uses a generic type for the labels contained in automata transitions, requiring only that some Boolean algebra (i.e. a class implementing an `IBooleanAlgebra` interface) be provided. The `IBooleanAlgebra` interface declares methods for conjuction, disjunction, negation and satisfiability checking over a given predicate type, as well as universally true/false predicate constants.

Therefore, the new ARMC tool contains the classes `SSA` and `SST` for simple symbolic automata and simple symbolic transducers, respectively. Similarly to `SFA` and `ST` in AutomataDotNet, these are wrapper classes for the core `Automaton` class, an instance of which is stored as a private attribute. Many methods then simply delegate to the underlying automaton.

The `SSA` and `SST` classes also provide a few methods necessary for ARMC which are not implemented in AutomataDotNet. For automata, these include collapsing an automaton based on some equivalence relation, as well as transforming an automaton to represent forward/backward state languages, forward/backward trace languages, bounded languages and combinations thereof. For transducers, additional methods implement transducer inversion, composition and union construction, as well as transforming an automaton by applying the transducer.

[1] Transducer composition is only implemented in the `ST` class, but it is not a difficult algorithm to reimplement.

`SSA` and `SST` also differ on the type of label used, as well as the associated Boolean algebra. The `SSA` class represent transition labels using the `Predicate` class, which consists of a type (*in* or *not in*) and a set of symbols. `SST` then uses the `Label` class, consisting of two `Predicate` instances for the input and output, as well a Boolean indicator of an identity label (identity labels do not use the output predicate). Note that ε is represented with a null value in place of a `Predicate` object.

The `PredicateAlgebra` implements the `IBooleanAlgebra` interface by providing methods for Boolean operations over `Predicate` instances. It stores a set of symbols (i.e. the alphabet) as a read-only public attribute (important for satisfiability checking). The `LabelAlgebra` has the same function for `Label` instances, as well providing a method for combining transducer labels.

It is also very practical to be able to load and save automata/transducers to files. Though AutomataDotNet does not provide implementations for this, methods for parsing and printing automata/transducers using text file formats are provided in the ARMC tool. The supported formats are based on the Timbuk [10], FSA [20] and FSM [17] libraries[2]. For printing, the DOT format is also supported, allowing image files to be created using the Graphviz tool [12].

Since the parsing and printing algorithms are very similar for both automata and transducers, the static classes `Parser` and `Printer` use an `ISSAutomaton` interface for accessing the structure of automata and transducers alike, as well as an `ILabel` interface which envelops both `Predicate` and `Label` types.

For clarification, Figure 4.1 depicts the relationships between the above described classes in a UML class diagram. Some of the more important attributes and methods are also listed, though not all. Note that all of these classes use a generic type for a symbol, meaning that one may choose to use integers instead of strings, for example[3]. Integers are used for states, though `SSA` and `SST` add the option of attaching names to the states for more convenience (as well as an option of naming the automaton/transducer).

4.4 Configuration

ARMC allows many different combinations of configuration settings. In order to avoid impractically long method parameter lists and command-line invocations, the new tool uses a single class, `Config`, to take care of the different options. In addition to the many public attributes, the class implements parsing and printing configurations to/from a text file. When the console application is invoked with a `-g` or `--generate-config` option, it will merely create a configuration file with default settings. The user may then modify the configuration file according to their needs before running the ARMC method. A simple "`key = value`" format is used in the file, and every setting is accompanied with comments detailing its syntax and semantics.

The three main settings are the files containing the SSA representing initial ($Init$) and property-violating configurations (Bad), and the SST representing the one-step transi-

[2] We support a version of Timbuk that is compatible with the symboliclib library [4].

[3] The console application uses strings, though. A different type is only possible for library usage.

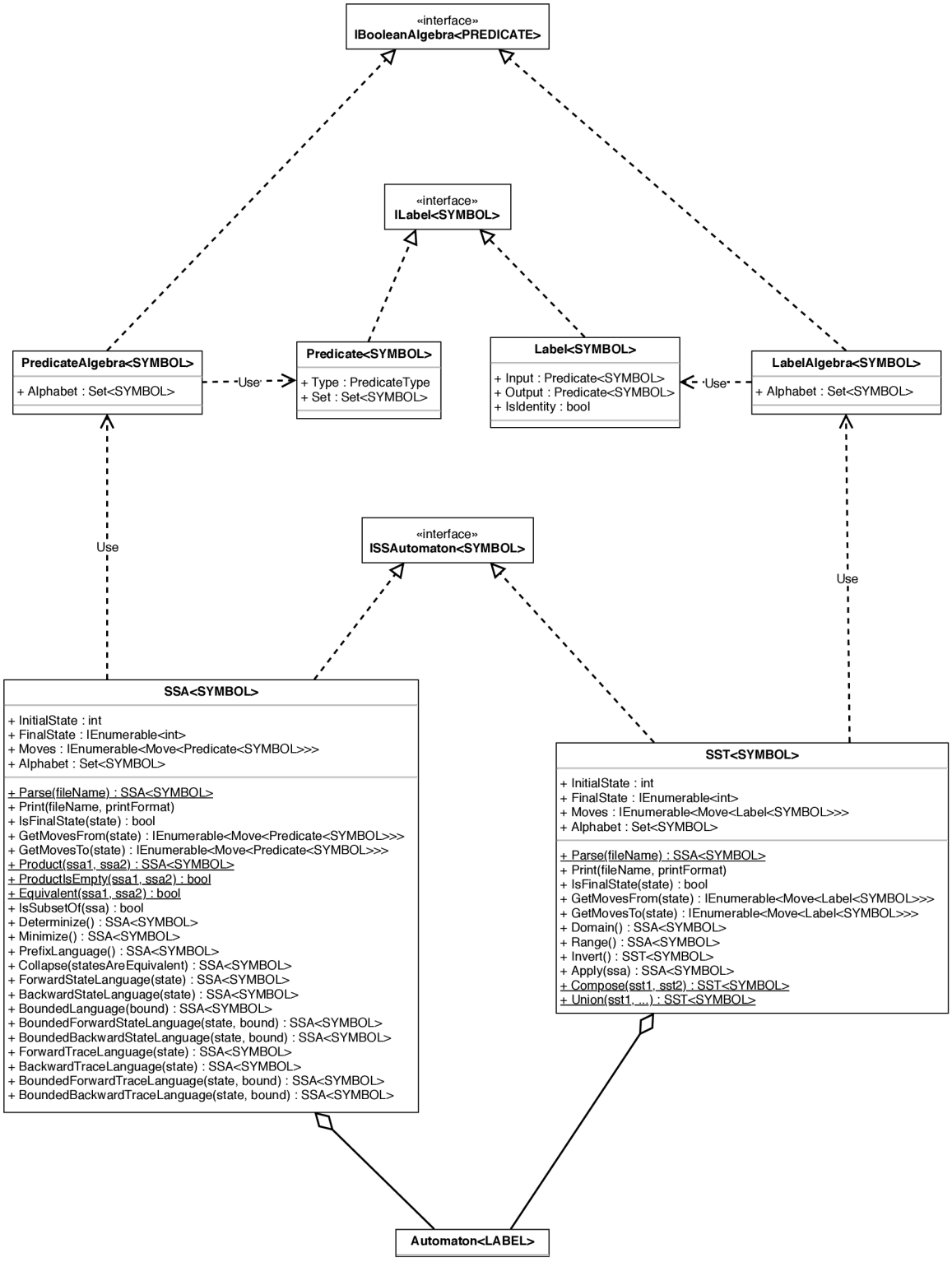


Figure 4.1: UML class diagram for simple symbolic automata/transducers.

tion (τ)[4]. These may also be specified using command-line arguments (using `-i`/`--init`, `-b`/`--bad` and `-t`/`--tau`), which override the configuration file.

The user may also specify a number of general settings. These include the direction of the computation[5] and whether forward or backward languages are used. The user may also specify a time interval after which ARMC will terminate with a "don't know" answer (given that termination is not guaranteed), and optionally enable verbose output, as well as if all created automata should be printed to files in a comprehensive directory structure, in which case the file format may also be selected (direct image creation is enabled if Graphviz is installed). All these files will be placed in a chosen output directory, which is also used for printing a counterexample if the property is violated.

The user also has the choice of which abstraction technique to use. If predicate languages are enabled, then one may also select if state languages of $Init$ or Bad should be used as initial predicates, whether to also include the domain or range of the specified transducer(s) among the initial predicates, and whether to enable a heuristic which considers only important states or one or two key states.

On the other hand, if the user selects finite length languages as their abstraction technique, then they again have a number of additional options to choose from. One may enable the use of trace languages, and decide on how to initialize and increment the bound n. The initial value may be equal to 1 or the number states in $Init$ or Bad (the number of states may also be halved). Similarly, the bound may be incremented by 1 or the number of states in M_k or X_k (which may again be halved).

4.5 Abstraction

Since ARMC uses one of two different abstraction and refinement techniques, the new tool handles this by providing two different implementations of the same interface. An abstract class `Abstraction` declares the basic methods for abstraction (i.e. collapsing automata) and abstraction refinement. A `Collapse` method transforms an automaton by collapsing its states, while a `Refine` method is used to update the abstraction (i.e. add predicate automata or increase bound) based on the automata M_k or X_k. In addition, a method called `StatesAreEquivalent` is also declared. It determines whether two specified states are equivalent within a specified automaton. This is used in a default implementation of `Collapse` as the equivalence relation.

The two subclasses are named `PredicateAbstraction` and `FiniteLengthAbstraction`. While the latter uses the default implementation for collapsing automata, the former overrides it in favour of a more efficient algorithm and an optional use of heuristics. Figure 4.2 illustrates the relationships between these classes with a UML class diagram.

[4] Multiple transducers may be specified, in which case their union forms the one-step transition.

[5] A forward computation checks that $\varrho_\tau{}^*(L(Init)) \cap L(Bad) = \emptyset$, whereas a backward computation checks that $\varrho_\tau^{-1}{}^*(L(Bad)) \cap L(Init) = \emptyset$.

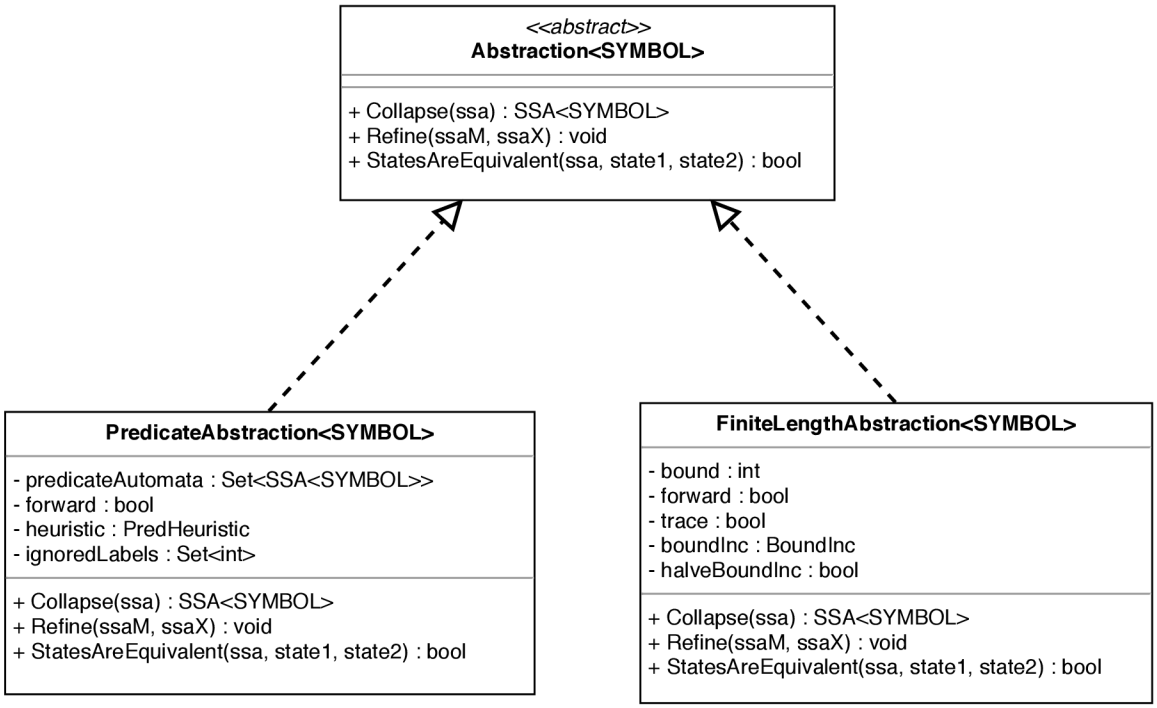


Figure 4.2: UML class diagram for predicate-based and length-based abstraction techniques.

4.6 The Main Method

Building on the components described in previous sections, the core class within the tool is called `ARMC`. The constructor takes a `Config` object as a parameter and uses it to set up the method. The constructor also throws an exception if the languages of $Init$ and Bad intersect, as that would cancel the need for ARMC to be performed at all.

The class contains two other public methods. `Verify` runs the entire algorithm (as depicted in Listing 3.1), returning a Boolean value indicating the verification result. If a property violation was found, then a counterexample is returned as an output parameter[6]. If a timeout is specified in the configuration, then the method throws an exception when the time limit is reached. Otherwise, the method is not guaranteed to terminate.

For this reason an alternative `VerifyStep` method is also provided for performing a single iteration of the outer loop (i.e. abstraction fixpoint computation and counterexample checking). It has the same type signature as `Verify`, except that it may return an empty value instead of a Boolean if the verification step was not conclusive (i.e. a spurious counterexample was found). The advantage of using this method is that it will terminate.

Note that the time limit is only checked in situations when there is no indication that a result may be near. `Verify` only checks it after each inconclusive verification step, while `VerifyStep` checks it after each round of the fixpoint computation. The reasoning for the latter is that the fixpoint computation is only guaranteed to be finite if τ was specified correctly (i.e. also represents an identity relation), and even it is finite, it might still take a long time.

[6] A method for printing this counterexample to a given directory is also provided.

Chapter 5

Implementation

This chapter covers the implementation of the new abstract regular model checking tool, based on its design as described in Chapter 4. In general, the source code is written in a functional programming style where suitable. Arrow functions and function delegates are often utilized, as are higher-order functions operating over enumerable data types (lists, sets, dictionaries, etc.) provided by .NET's Language Integrated Query (LINQ) framework. For handling command-line arguments, the implementation makes use of the open-source options parser *NDesk.Options* by Jonathan Pryor [21].

First, Sections 5.1 and 5.2 describe the algorithms used to provide some necessary capabilities missing in AutomataDotNet, for automata and transducers, respectively. Section 5.3 then covers the implementation of parsing and printing of automata/transducers in the various supported formats, as well as the ARMC configuration (using a more maintainable reflection-based approach). Finally, Section 5.4 describes an efficient algorithm used for collapsing automata based on predicate languages, while also providing some implementation notes on the two optional heuristics.

5.1 Automata Algorithms

Depending on the ARMC configuration used, it may be required to transform an automaton M to an automaton M' which for a given state q accepts its forward/backward (trace) language, possibly of a bounded word length n. For a forward state language, it suffices to replace the initial state with q, while a backward state language entails replacing the set of final states with $\{q\}$. Trace language computation is delegated to an AutomataDotNet method for computating a prefix language (ε-transitions are added from each state to a new final state). For bounded languages, one may construct a product automaton such that $L(M') = L(M) \cap L(M_\Sigma^{\leq n})$, where the automaton $M_\Sigma^{\leq n} = (\{0, \dots n\}, \delta_\Sigma, \{(i, \top) \to \{i+1\} \mid 0 \leq i < n\}, 0, \{0, \dots n\})$ accepts all $w \in \Sigma^*$ such that $|w| \leq n$. Related constructions (e.g. bounded forward trace languages) are obtained by a composition of the algorithms above.

As for collapsing an automaton based on an equivalence relation, an algorithm for this is presented in Listing 5.1. Every state is mapped to a representative of its equivalency class by use of a dictionary. Each new state is tested for equivalency with each representative in order to determine which equivalency class it belongs to. If none is found, the state

```
Collapse(M = (Q, 𝔖_Σ, Δ, q_0, F), ~ ⊆ Q × Q)
begin
    m : Q → Q; m := ∅ # maps state to equivalency class representative
    for ∀q ∈ Q do
        for ∀q* ∈ ran(m) do
            if q ~ q* then
                m := m ∪ {q → q*} # add to equivalency class
                break
            endif
        endfor
        if q ∉ dom(m) then
            m := m ∪ {q → q} # new equivalency class
        endif
    endfor
    Q_/~ := ran(m)
    let q²_/~ ∈ Δ_/~(q¹_/~, ψ) ⟺ m⁻¹(q²_/~) ∈ Δ(m⁻¹(q¹_/~), ψ)
    q⁰_/~ := m(q_0)
    F_/~ := {m(q_F) | q_F ∈ F}
    return M_/~ = (Q_/~, 𝔖_Σ, Δ_/~, q⁰_/~, F_/~)
end Collapse
```

Listing 5.1: Pseudocode for collapsing a SSA M according to an equivalence relation $\sim$.

becomes the representative of a new equivalency class. The resulting quotient automaton is then obtained by translating all the states according to this dictionary.

The implementation of the `SSA` class also includes a workaround for an inconvenience presented by AutomataDotNet. For any methods working with two automata (e.g. product construction), it is first checked that the `IBooleanAlgebra` objects are identical. For `PredicateAlgebra` objects this is unnecessarily strict, as they only need to operate over the same alphabet in order to be compatible. The solution is for the `SSA` class to maintain a static attribute which is a dictionary mapping an alphabet (i.e. a set of symbols) to a `PredicateAlgebra` instance which uses that alphabet. The constructor then checks if this dictionary already contains en entry for the given alphabet, in which case the correspoding algebra object is used.

5.2 Transducer Algorithms

A central method required for ARMC is to transform an automaton M to another automaton M' using the transducer τ such that $L(M') = \varrho_\tau(L(M))$. The basic idea is to use a product construction for the states of τ and M, but replace each predicate with the output of the transducer label whose input satisfies that predicate. The situation is then also slightly complicated by the appearance of ε and ι among these labels. The algorithm is illustrated in Listing 5.2.

The algorithm for composing transducers is the same as described in 2.2.4. Inverting a transducer consists of modifying the transitions by swapping the input and output of each label, with the exception of identity labels, which are not changed.

```
Apply(τ = (Q_τ, Σ_Σ, Δ_τ, q_0^τ, F_τ), M = (Q_M, Σ_Σ, Δ_M, q_0^M, F_M))
begin
    M := RemoveEpsilons(M)
    Δ' := ∅
    S ⊆ Q_τ × Q_M; S := {(q_0^τ, q_0^M)}
    m : Q_τ × Q_M → ℕ_0; m := {(q_0^τ, q_0^M) → 0} # maps pair of states to identifier
    i := 1
    while S ≠ ∅ do
        select (q_τ, q_M) ∈ S; S := S \ {(q_τ, q_M)}
        D := ∅ # tuples with predicate and target state pair
        for ∀D_τ = Δ_τ(q_τ, ψ_1, ψ_2) do
            if ψ_1 = ε then
                if ψ_2 = ι then φ' := ε
                        else φ' := ψ_2
                endif
                D := D ∪ {(φ', D_τ × {q_M})}
            else
                for ∀D_M = Δ_M(q_M, φ) do
                    if [[ψ_1 ∧ φ]] ≠ ∅ then # satisfiability check
                        if ψ_2 = ι then φ' := ψ_1 ∧ φ
                                else φ' := ψ_2
                            D := D ∪ {(φ', D_τ × D_M)}
                        endif
                    endif
                endfor
            endif
        endfor
        for ∀q' ∈ ⋃_{(φ',D')∈D} D' do
            if q' ∉ dom(m) then # add new state pair
                S := S ∪ {q'}
                m := m ∪ {q' → i}
                i := i + 1
            endif
        endfor
        Δ' := Δ' ∪ {(q, φ') → D'_m | (φ', D') ∈ D ∧ D'_m = {m(q') | q' ∈ D'}
    endwhile
    Q' := ran(m)
    q'_0 := m(q_0^τ, q_0^M)
    F' := {q' | m(q_τ, q_M) = q' ∧ q_τ ∈ F_τ ∧ q_M ∈ F_M}
    return M' = (Q', Σ_Σ, Δ', q'_0, F')
end Apply
```

Listing 5.2: Pseudocode for applying a SST τ to a SSA M.

For creating a union of transducers, one must create a new initial state and ε/ε-transitions to each transducer's initial state. All states and transitions are then merged, but first it must be ensured that their sets of states are disjoint. This is done by translating each transducer's states such that the first transducer's states cover the range $\{1,\dots n_1\}$, the second's cover $\{n_1+1,\dots n_1+n_2\}$, and so on (n_i is equal to the number of states of the i-th transducer). 0 is reserved for the new initial state. The union method takes an arbitrary number of transducer parameters, in order to allow for a single new initial state to be added instead of creating a new one for each pair.

5.3 Parsing and Printing

As described in Section 4.3, a joint interface allows the same algorithm to be used for parsing and printing both automata and transducers. The interface declares the presence of a public attribute used to differentiate automata from transducers. This is usually used to determine how to parse/print transition labels, which is the main point where the supported file formats differ. The implementation makes use of the joint interface for automata and transducer labels.

Parsing is performed with the use of C#'s powerful regular expression utilities. The main parsing method must first determine the type of file format being used. If the file extension gives no indication, then the format is guessed based on the structure of the file contents.

When printing in the FSA format (in fact valid Prolog code), the components are formatted and commented in order to be more readable. When the DOT format is used, HTML formatting and unicode characters are used to make the resulting graphs more clear (symbols are typeset in italics, and state names allow for the use of subscripts). Since sets appearing in predicates may be very large and result in labels overlapping (and thus becoming incomprehensible), line breaks are added at regular intervals between groups of symbols within a set.

Note that because of the limits imposed by the various formats on which characters may appear among symbols, an automaton may change when printing and parsing in different formats.

As far as the printing and parsing of `Config` instances is concerned, the implementation makes use of C#'s reflection toolkit in order to make the code more maintainable. As the class contains a large number of public attributes, they may all be fetched in a loop and their values handled according to their type. Comments are also attached to each attribute by use of a C# feature called *custom attributes*. All of this means that in order to add or modify configuration settings, the programmer need only make the change in the attribute declaration, and the methods for printing and parsing will adjust themselves accordingly without any attention needed. While this approach increases maintainability, it is also a little less efficient. However, since the configuration is only parsed/printed once for each ARMC run, this is less of a concern.

5.4 Optimized Predicate-Based Abstraction

The definition of the $\mathbb{F}_{\mathcal{P}}$ and $\mathbb{B}_{\mathcal{P}}$ schemas based on predicate languages in 3.3.3 indicates that for every refinement, an automaton for each state language is added to the set of predicate automata $\mathcal{P}$. This implementation is unnecessarily ineffecient, as the automata accepting languages of states from the same automaton would share large parts of their structure. A better option is to simply add the X_k automaton as is, and consider the languages of all its states as separate predicates when collapsing.

Given that the collapsing of an automaton M consists of comparing intersections of all its states with all the states from predicate automata, it is good to have an efficient algorithm for deciding this. The basic idea is to label every state of M with a set of predicate automata states with whose forward/backward languages they have a non-empty intersection. For the

$\mathbb{F}_{\mathcal{P}}$ schema, this may be done with backward synchronous product construction. Specifically, for each predicate automaton P, every final state of M is first labelled with the final states of P. Then if a state q_2^M is labelled with q_2^P and there exist predicates ψ and φ such that $q_2^M \in \Delta_M(q_1^M, \psi)$, $q_2^P \in \Delta_P(q_1^P, \varphi)$ and $\psi \wedge \varphi$ is satisfiable, q_1^M is labelled with q_1^P. Similarly, a forward synchronous product construction is used for the $\mathbb{B}_{\mathcal{P}}$ schema, with the difference being that we begin by labelling the initial state of M with initial state of each predicate automaton, and then we follow the transitions from source state to target states. After the labelling is done, the equivalence of two states in M consists of checking that their sets of labels are equal.

One detail that the implementation need take care of, given that states are represented by integers, is ensuring that the predicate automata use disjoint sets of integers for their states. Therefore, whenever a new predicate automaton X_k is to be added as part of the abstraction refinement, its state numbers are translated in order to form a sequence $n, \ldots, n+m-1$, where m is the number of states in X_k and n is total number of states of all the preceding predicate automata[1].

The `PredicateAbstraction` class also maintains a set of integers denoting labels to be ignored. This is used for the heuristics described in 3.3.3 which only consider important or key states when collapsing automata. Once every state has been labelled (resulting in a dictionary mapping an integer to a set of integers), all ignored labels are removed using set subtraction. Only then is the collapsing algorithm performed.

For the *important tail/head part* heuristic, the labelling is performed after adding the new predicate automaton, and each of its states appearing among the labels is collected in a set of important states. All non-important states are then added to the set of ignored labels.

The *key states* heuristic also computes the set of important states. It then tries to find one key state among them which will suffice to prevent X_k from intersecting with the collapsed version of M_k when ignoring all other states of X_k (they are temporarily added to the ignored labels and subsequently removed if the intersection is non-empty). If one such state cannot be found, the algorithm attempts to find two key states, falling back on the important states heuristic if this also fails.

[1] Before this, the new predicate automaton has its ε-transitions removed in order to simplify the labelling algorithm.

Chapter 6

Experiments

This chapter describes the various verification tasks which were used in testing the new abstract regular model checking tool. Each section describes an algorithm and some property which we verify, the models used to that effect, and the experimentation results. Most of the algorithms are concerned with process synchronization, where we verify a safety property, specifically whether they guarantee mutual exclusion (i.e. no more than one process may be in a critical section at any given time). We also include examples of a push-down system and a system with a queue, where we verify that some actions happen in the correct order.

The presented times were obtained on a computer with a 2.5 GHz Intel Core i5 processor, and do not include the time needed for I/O operations. The effectiveness of the computation varied greatly based on the configuration used. For all of the modeled algorithms, results ranged from reaching a fixpoint in less than a second without needing to perform any refinements to computations so time-consuming that they had to be terminated prematurely (and in some cases were clearly never going to finish, typically when using trace languages in a backward computation).

Section 6.1 describes the *bakery algorithm*. Since this is one of the simpler algorithms, the models and experimenation results are described in more detail than for other algorithms, in order to give a clear idea of the modeling techniques used and the types of effects different configurations may have. Other process synchronization algorithms are covered in Sections 6.2 (*Dijkstra's algorithm*), 6.3 (*Burns' algorithm*) and 6.4 (*Szymański's algorithm*). For the latter, we also use an erroneous version to demonstrate the generation of a counterexample. Section 6.5 then describes an example of a simple push-down system that uses recursive procedures, before Section 6.6 covers the *alternating bit protocol*. Finally, Section 6.7 summarizes the results from previous sections, and also shows a rough speed comparison with one of the ARMC prototypes.

6.1 Bakery Algorithm

The bakery algorithm by Leslie Lamport is used to ensure mutual exclusion between multiple processes trying to enter a critical section [13]. It is inspired by the analogy of a bakery where upon entrance, every new customer receives a ticket from a numbering machine which

is higher by 1 than the previously issued ticket, and is only served once his ticket number is the lower than that of all other waiting customers. A customer represents a process, while a customer being served represents a process being in the critical section.

Each process is identified by the number i. A global array e of Boolean values (all initially false) determines if a given process is entering the "bakery" (i.e. obtaining its ticket number), while a global array t of non-negative integers (all initially 0) determines each process' ticket number. An important condition is that there is no upper limit on the possible ticket numbers. If $t[j] = 0$, it means that the process j does not currently have a ticket.

Due to the limitations of computer architecture, it is possible that multiple processes may receive the same ticket number. In this case, the process with the lowest identifier i has a higher priority with regards to entering the critical section. However, as we will shortly show that our model does not allow for duplicate ticket numbers, we shall ignore this and restrict ourselves to a simplified version of the Bakery algorithm, as shown in Listing 6.1 in the form of pseudocode for a process i. A process finds itself in the critical section when it is at line number 5.

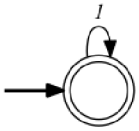
```
1 | e[i] := true
2 | t[i] := 1 + t[j]: ∀k: t[j] ≥ t[k]
3 | e[i] := false
4 | await ∀j ≠ i: ¬e[j] ∧ (t[j] = 0 ∨ t[i] < t[j])
5 | t[i] := 0; goto 1
```

Listing 6.1: Bakery algorithm.

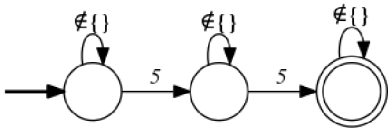
We model this algorithm by representing a process state with a letter, which corresponds to the line number from Listing 6.1 that the process is about to execute. A word is then made up of the states of each process, of which there may be an arbitrary amount. Since the range of ticket numbers is unbounded, the values of the array t cannot represented by a process state. Instead, the letters in a word are ordered ascendingly from left to right according to the process' ticket number (it is for this reason that duplicate ticket numbers are not possible). The value of e is easily discernable from each process' state, as it is always true for lines 2 and 3, and false otherwise.

Figure 6.1 shows the models we used for verifying the mutual exclusion property of Lamport's algorithm. All processes start at line 1 (see 6.1a). The property is violated when there are two or more processes in the critical section, i.e. at line 5 (see 6.1b). The transducer shown in 6.1c is the union of the identity relation (i.e. no process does anything) and all possible steps between lines of a single process. Most steps are trivial, meaning only a given process' letter changes while all others are copied. One exception is the transition from line 2 to 3, which must also move the process' letter to the end of the word. The other exception is the transition between lines 4 and 5 (entering the critical section). In this case, all the other processes must satisfy the condition that they are not obtaining a ticket and either have no ticket or have a higher ticket number. This means that all preceding letters ($t[j] < t[i]$) must satisfy $\neg e[j] \wedge t[j] = 0$ (i.e. $(\notin, \{2,3\}) \wedge (\in, \{1\}) = (\in, \{1\})$), while successive letters ($t[i] < t[j]$) must satisfy $\neg e[j]$ (i.e. $(\notin, \{2,3\})$).

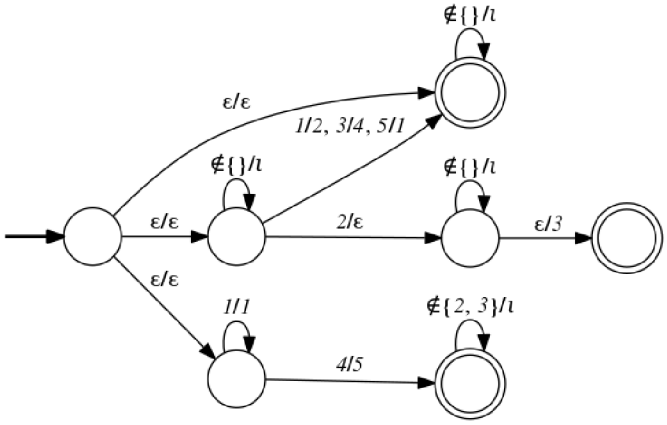
When verifying the bakery algorithm using our new ARMC tool, a forward computation coupled with the $\mathbb{F}_{\mathcal{P}}$ schema proved to be the fastest. The fixpoint depicted in Figure 6.2a was reached after 6 steps (7 if the domain or range were among the initial predicates). When the Bad automaton was used as the initial predicate (with or without $Init$), this resulted



(a) The initial configurations.



(b) The bad configurations.

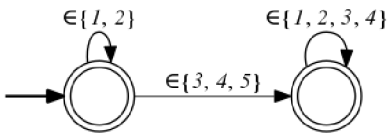


(c) The transition.

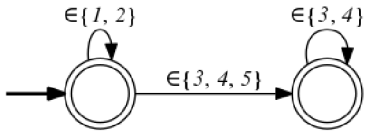
Figure 6.1: Models for the bakery algorithm.

in the fastest time of 0.07 seconds and no refinements were necessary (one refinement was needed when Bad was omitted). Other forward computations also succeeded in verifying the mutual exclusion property, reaching the more precise fixpoint depicted in Figure 6.2b after two refinements or fewer. For the $\mathbb{B}_n^L$ schema, it was crucial to use a bound increment of 1.

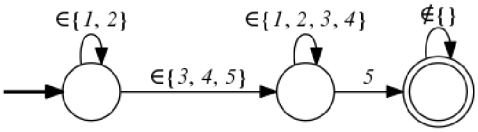
For length-based abstraction, a backward computation coupled with the $\mathbb{F}_n^L$ schema was fastest (0.10 s). The fixpoint reached using a backward computation was invariably that depicted in Figure 6.2c. The $\mathbb{B}_\mathcal{P}$ schema required the inclusion of Bad or the domain or range of τ among the initial predicates. Other backward computations failed to terminate entirely — the $\mathbb{B}_n^L$ schema proved too precise, while the schemas based on trace languages could never distiguish between any of the states of Bad.



(a) Forward computation fixpoint ($\mathbb{F}_\mathcal{P}$ schema).



(b) Forward computation fixpoint ($\mathbb{F}_n^L$, $\mathbb{B}_n^L$, $\mathbb{F}_n^T$, $\mathbb{B}_n^T$ and $\mathbb{B}_\mathcal{P}$ schemas).



(c) Backward computation fixpoint.

Figure 6.2: Various fixpoints for bakery algorithm.

6.2 Dijkstra's Algorithm

Listing 6.2 shows an idealized version of Dijkstra's mutual exclusion protocol from [19]. Each process has an identifier i, and has access to a global array f of flags ranging over $\{0,1,2\}$ (initially all 0) and a global process identifier p. Line number 7 represents the critical section.

```
1 | f[i] := 1
2 | if p ≠ i then goto 3 else goto 5
3 | await f[p] = 0
4 | p := i
5 | f[i] := 2
6 | if ∃j ≠ i: f[j] = 2 then goto 1 else goto 7
7 | f[i] := 0; goto 1
```

Listing 6.2: Dijkstra's algorithm.

We model a configuration as a word where each letter represents the state of one process. The state of a process i is a made up of the current line number, the value of $f[i]$ and a Boolean indicating if $p = i$ (this always holds for precisely one process). Initially, all processes are at line 1, f contains only zeros, and for one arbitrary process the $p = i$ flag holds, while not holding for any of the other processes. A property violation is modeled as two or more processes being at line 7 (with any flag values). The transducer is again formed as the union of an indentity relation and the various steps one process may take to change lines. A branching statement is modeled by progressing according to the *if*-branch if the condition holds, while progressing according to the *else*-branch if the negation of the condition holds. When the condition being tested uses an existential quantifier (e.g. when stepping from line 6 to line 1), the transducer contains two separate paths differing on whether this other process is found before or after the active process in the word ordering.

By far the most efficient ARMC configuration for verifying Dijkstra's algorithm proved to be a predicate-based abstraction combined with a forward computation using Bad as the initial predicate. The fixpoint was reached in 8 steps without refinements, with the $\mathbb{B}_{\mathcal{P}}$ schema recording the fastest time (0.23 s). If $Init$ was used instead of Bad, the computation took about a hundred times longer (and required the use of the key states heuristic for the $\mathbb{B}_{\mathcal{P}}$ schema). For length-based abstraction, a forward computation combined with forward (trace) languages was the only successful method, with the $\mathbb{F}_n^L$ schema recording the faster of the two times (9.15 s). Backward computations proved entirely unsuccessful for this algorithm.

6.3 Burns' Algorithm

Listing 6.3 illustrates Burn's mutual exclusion algorithm [16]. Each process has access to a global array f of Boolean values (all initially false). Line number 6 represents the critical section.

In our model, the state of a process i is a pair made up of the line number and the value of $f[i]$. The same modeling techniques as for previous algorithms are used here.

Predicate-based abstraction was considerably faster in verifying Burns' algorithm. The fastest time was achieved with a forward computation using Bad as a predicate, when a

```
1 | f[i] := false
2 | if ∃j < i: f[j] then goto 1 else goto 3
3 | f[i] := true
4 | if ∃j < i: f[j] then goto 1 else goto 5
5 | await ∀j > i: ¬f[j]
6 | f[i] := false; goto 1
```

Listing 6.3: Burns' algorithm.

fixpoint was reached in 8 steps without refinements (0.8 s). Using $Init$ proved to be almost as fast, and backward computations using the $\mathbb{F}_{\mathcal{P}}$ schema were not much slower either (one refinement was needed in these cases). The combination of a backward computation with the $\mathbb{B}_{\mathcal{P}}$ schema was not successful.

For finite-length languages, the fastest time was achieved with a forward computation coupled with the $\mathbb{B}_n^T$ schema (0.63 s). Using the $\mathbb{F}_n^L$ or $\mathbb{F}_n^T$ schemas instead proved to be only a little slower. A backward computation using the $\mathbb{F}_n^L$ schema was also successful, though much more computation time was needed. In each of these cases, the bound had to equal 1, and a fixpoint was reached after 13 or 14 steps without the need for any refinements. None of the other combinations were successful.

6.4 Szymański's Algorithm

A more complex mutual exclusion algorithm was devised by Boleslaw Szymański, whose advantages include linear wait [23]. Listings 6.4 shows a (slightly idealized) version of this algorithm as described in [25]. The algorithm uses an array pc which holds the current program counter (i.e. line number) of each process. It also uses the arrays s and w of Boolean values. Line number 7 represents the critical section.

```
1 | await ∀j ≠ i: ¬s[j]
2 | w[i] := true; s[i] := true
3 | if ∃j ≠ i: pc[j] ≠ 1 ∧ ¬w[j] then s[i] := false; goto 4 else w[i] := false; goto 5
4 | await ∃j ≠ i: s[j] ∧ ¬w[j] then w[i] := false, s[i] := true
5 | await ∀j ≠ i: ¬w[j]
6 | await ∀j < i: ¬s[j]
7 | s[i] := false; goto 1
```

Listing 6.4: Szymański's algorithm.

In order to model this protocol, we use $pc \times s \times w$ as our alphabet, i.e. a letter representing the state of some process i is a tuple $(pc[i], s[i], w[i])$ (the first part ranges over $\{1, \ldots 7\}$, while the latter two are either true or false). Initially, the arrays s and w hold arbitrary values.

When predicate languages were used for the verification of Szymański's algorithm, only forward computations were successful. The inclusion of the domains and/or ranges of the individual transducers (whose union forms the one-step transition) among the initial predicates helped speed up the computation significantly. The $\mathbb{F}_{\mathcal{P}}$ schema proved fastest (needing no refinements) when the transducer ranges were included, with the fastest time recorded when these were added to Bad to form the initial predicates (0.13 s). The $\mathbb{B}_{\mathcal{P}}$

schema worked best when the transducer domains were included as initial predicates, with 4 refinement steps being required. Finite-length languages were only successful when used in a forward computation using forward (trace) languages. The $\mathbb{F}_n^T$ schema yielded the fastest time after 13 steps without refinement (2.49 s).

6.4.1 A Faulty Version of Szymański's Algorithm

While testing the new tool, we discovered an error in the pseudocode for Szymański's algorithm presented by [19]. The incorrect version is shown in Listing 6.5. It is the same as the correct version above, with the exception of lines 3 (where the condition contains $pc[j] \neq 2$ instead of $\neg w[j]$) and 6 (where the condition is weakened by adding $\neg w[j]$). These changed lines are highlighted.

```
1 await ∀ j ≠ i: ¬s[j]
2 w[i] := true; s[i] := true
3 if ∃j ≠ i: pc[j] ≠ 1 ∧ pc[j] ≠ 2 then s[i] := false; goto 4 else w[i] := false; goto 5
4 await ∃j ≠ i: s[j] ∧ ¬w[j] then w[i] := false, s[i] := true
5 await ∀ j ≠ i: ¬w[j]
6 await ∀ j < i: ¬s[j] ∨ ¬w[j]
7 s[i] := false; goto 1
```

Listing 6.5: Incorrect version of Szymański's algorithm.

The new ARMC tool was used to find a counterexample showing how two or more processes could reach the critical section at the same time. Allowing for some differences in the precision of the abstraction, the discovered path (made up of 11 steps) was always the same[1]. It is shown in Figure 6.3, where we suppose two processes for simplicity (there may be an arbitrary number of other processes which stay at line 1) with the s and w arrays initially containing only false values. While this path has the process with a higher identifier progressing faster, it is also possible for the paths of the two processes to be swapped (in which case the w array may sometimes contain arbitrary values initially). Note that the change to line 3 is what enables the fourth step, while the change to line 6 enables the last or second-to-last step (depending on which process progresses first). We also tried correcting only one of the incorrect lines, which in both cases also resulted in a counterexample being found after 11 steps (though there were less alternative branches).

Only forward computations succeeded in uncovering the counterexample (backward computations would not terminate). When using predicate languages, the inclusion of the domains and/or ranges of the transducers meant that no refinements where necesarry. The $\mathbb{B}_{\mathcal{P}}$ schema proved faster, with the fastest time being recorded using $Init$, Bad and the domain and range of each transducer (1.18 s). For finite length languages, the $\mathbb{F}_n^T$ schema was fastest (3.76 s). The $\mathbb{B}_n^L$ schema was much slower than the others (it was the only one to require refinements), even when speeded up significantly by incrementing the bound by just 1.

[1] The best way of interpreting the counterexamples generated by ARMC is by inspecting the X_i automata.

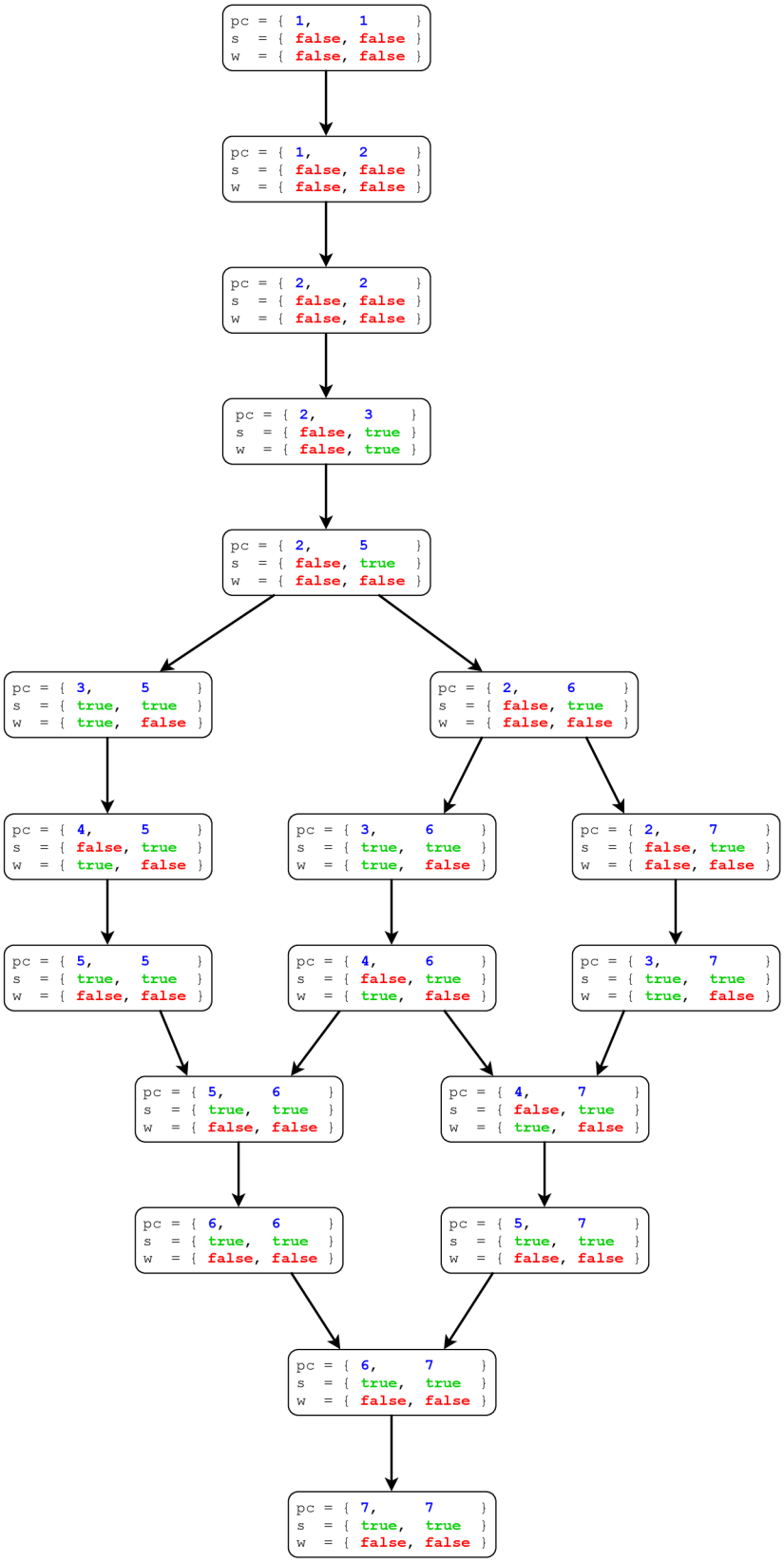


Figure 6.3: Counterexample to incorrect version of Szymański's algorithm.

6.5 Push-Down System

As a simple example of a push-down system using recursive procedures, we consider the *plotter* example from [9]. The plotter makes use of the commands `go_up`, `go_down` and `go_right` to draw a random bar chart. The algorithm is illustrated in Figure 6.4. It is required that no upward movement is immediately followed by a downward movement or vice versa.

We model this push-down system by having words contain the contents of the stack, which is made up of invoked procedures, each of which is marked by the flowchart node (see 6.4b) the procedure currently finds itself at. In order to verify the correctness property, the second half of the word lists all sequence movements in order of execution (the two parts are delimited by a special letter). Only the procedure at the top of the stack (leftmost letter) is executed. If a movement is generated, is it is appended to the end of the word. If a procedure is invoked, it is pushed to the top of the stack. For example, we transition from the configuration $s_2 m_4 s_5 a_1 | U$ to $s_4 m_4 s_5 a_1 | UU$, and then to $m_0 s_5 m_4 s_5 a_1 | UU$ (a refers to `main`).

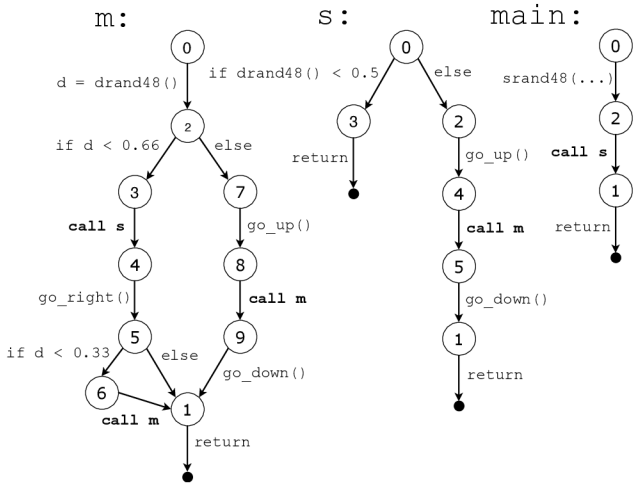
Only predicate-based abstraction succeeded in verifying the correct movement ordering of the plotter. Forward computations were faster in general, particularly when the transducer domain and/or range was included. The inclusion of the transducer range was crucial when using the $\mathbb{B}_{\mathcal{P}}$ schema. The fastest time was obtained when using the $\mathbb{F}_{\mathcal{P}}$ schema with Bad, $Init$, and the transducer range as the initial predicates, when 26 steps were performed without refinements (0.81 s). The only other successful combination was that of a backward computation with the $\mathbb{F}_{\mathcal{P}}$ schema, though it took around ten times longer than the other configurations on average.

```
void m() {
    double d = drand48();
    if (d < 0.66) {
        s(); go_right();
        if (d < 0.33) m();
    } else {
        go_up(); m(); go_down();
    }
}

void s() {
    if (drand48() < 0.5) return;
    go_up(); m(); go_down();
}

main() {
    srand48(time(NULL));
    s();
}
```

(a) Program.



(b) Flowchart.

Figure 6.4: Plotter example.

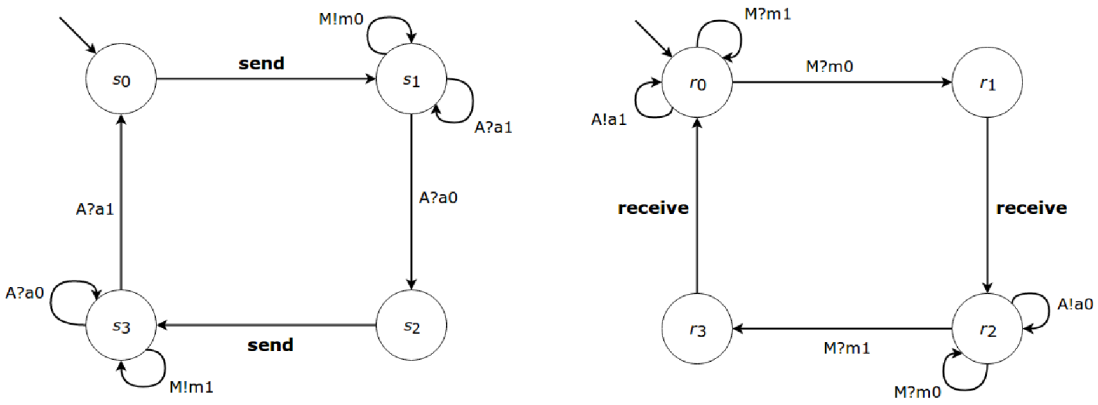
6.6 Alternating Bit Protocol

The alternating bit protocol is a simple network protocol operating at the data link layer, where messages are delivered over a lossy channel [2]. There are two FIFO channels, one for messages and the other for acknowledgements, both of which may lose messages but not reorder them. A single bit is used to ensure that the messages are delivered in the correct order. The sender sends a message with the sequence bit 0, and waits to receive an acknowledgement with same sequence bit. This is then repeated, but the sequence bit is inverted each time. Messages and acknowledgements may be resent, and those with incorrect sequence bits are ignored.

Figure 6.5 illustrates this protocol using automata for both the sender and receiver. The *send* and *receive* operations indicate calls from the upper layers of the protocols. M denotes the channel for messages, while A denotes the channel for acknowledgements. *M!m0* symbolizes writing a message with sequence bit 0 to the M channel, whereas *A?a1* symbolizes reading an acknowledgement with sequence bit 1 over from the A channel.

We model configurations of the alternating bit protocol as words made up of four separate parts (delimited by special letters). The first contains two letters denoting the state of the sender and receiver, respectively. The second and third parts contain the contents of the two lossy channels for messages and acknowledgements, respectively. The final part is where we log the upper layer *send* and *receive* calls, and this is where we verify one of the properties of this protocol — that these two operations alternate such that neither of them occurs consecutively. To simulate the fact that the two channels are lossy, the transducer contains transitions such that the second and third word parts may have messages/acknowledgements erased arbitrarily. In addition to this and the identity relation, the other transitions correspond to activities of either the sender or the receiver. These actions entail read and write operations on a channel (which take a leftmost or add a rightmost message to the queue) or invoking upper layer calls (which add to the end of the word), as well as modifying the sender/receiver states accordingly (by rewriting one of the first two letters).

As an example, consider the configuration $s_1r_1\#M_0|A_1A_1\$S$, where the receiver has just read a message with the sequence bit 0. It may then send an acknowledgement back to the sender by transitioning to $s_1r_2\#M_0|A_1A_1\$SR$ and then to $s_1r_2\#M_0|A_1A_1A_0\$SR$. The



(a) Sender.

(b) Receiver.

Figure 6.5: Alternating bit protocol.

next configuration may then be $s_1 r_2 \# M_0 | A_1 A_0 \$ SR$, either because of the acknowledgements channel being lossy, or because the sender has read an acknowledgement with sequence bit 1 and ignored it.

Only predicate-based abstraction techniques were successful in verifying the alternating bit protocol's message ordering. While a backward computation succeeded when coupled with the $\mathbb{F}_{\mathcal{P}}$ schema, forward computations were faster in general, especially when the transducer's domain and range were included among the initial predicates. Including Bad as a predicate too and using the $\mathbb{B}_{\mathcal{P}}$ schema resulted in the fastest computation time, when a fixpoint was found in 17 steps without the need to refine (0.38 s).

6.7 Summary

We summarize the results of our experiments described above in Table 6.1. The fastest time (in seconds), and the precise configuration leading to it, are shown separately for both abstraction techniques. We first note whether it was a forward or backward computation and the schema used. For predicate-based abstraction, this is followed by the initial predicates, and an optional heuristic. For length-based abstraction, we sometimes note the initial bound and its increment if necessary.

Additionally, we used the experimental results described in [6] to compare the speed of the new tool with the Prolog prototype[2]. The authors list the best scenarios and times for both abstraction techniques, so we used the same configurations when running our new tool, as well as the same models [14]. Table 6.2 shows the results for predicate-based abstraction, while Table 6.3 shows results for length-based abstraction. Our tool was usually slower, with a few notable exceptions when predicate languages were used. For finite length languages, we do not include results for the (correct) Petri net model of the readers–writers problem (the authors had to specially select the ideal bound) and Szymański's algorithm (which did not terminate for the given configuration, but finished using a different one in a faster time).

In general, while the referenced prototype shows a lot of variance between results for both abstraction techniques, our tool appears to run consistently faster when predicate-based abstraction is used instead of length-based abstraction.

[2]Since the prototype's results were obtained on a different machine, this is only a rough comparison.

Experiment	$\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$	Time	$\mathbb{F}^L_n/\mathbb{B}^L_n/\mathbb{F}^T_n/\mathbb{B}^T_n$	Time
Bakery	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.07	Bw, $\mathbb{F}^L_n$	0.10
Dijkstra	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.23	Fw, $\mathbb{F}^L_n$	9.15
Burns	Fw, $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.08	Fw, $\mathbb{B}^T_n$	0.63
Szymański	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\|Act]$	0.84	Fw, $\mathbb{F}^T_n$	2.49
Szymański (faulty)	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\|Init\|Grd\|Act]$	1.18	Fw, $\mathbb{F}^T_n$	3.76
PDS	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad\|Init\|Act]$	0.81		
ABP	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\|Init\|Grd\|Act]$	0.38		

Table 6.1: Summary of results.

Experiment	Configuration	Prototype	New Tool
Bakery	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02	0.06
Bakery – communal liveness	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad \vert Grd]$	0.13	0.12
Bakery – individual liveness	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$, key st.	19.41	71.21
ABP	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init \vert Grd]$	0.68	0.39
Burns	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.06	0.10
Dijkstra	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.73	2.50
PDS	Bw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02	0.13
Petri net: readers–writers	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad \vert Grd]$	5.86	2.16
Petri net: readers–writers (faulty)	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init \vert Grd]$	0.81	0.29
Szymański	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Init \vert Grd]$	0.55	5.66
List reversal	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad \vert Grd \vert Act]$	1.29	0.12

Table 6.2: Speed comparison with prototype using predicate-based abstraction.

Given our main aim, the slower computation speeds are not a great concern. Our focus was on a good design and clean implementation, in order to provide a maintainable tool with a convenient interface, and we have therefore avoided premature optimization. Though the tool's modularity (e.g. using an external library) is useful for maintainability purposes, it may come with the cost of a computational overhead. Moreover, the prototype may include optimizations that were not described in the referenced article.

Experiment	Configuration	Prototype	New Tool
Bakery	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert/2$	0.02	0.08
Bakery – communal liveness	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert$	0.14	0.67
Bakery – individual liveness	Fw, $\mathbb{F}_n^T$, 1	8.66	122.63
ABP	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.32	2.72
Burns	Fw, $\mathbb{B}_n^T$, 1	0.31	0.60
Dijkstra	Fw, $\mathbb{F}_n^T$, 1	1.75	7.19
PDS	Bw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.02	0.09
Petri net: readers–writers (faulty)	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert$	0.73	2.51
List reversal	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.61	1.11

Table 6.3: Speed comparison with prototype using length-based abstraction.

Chapter 7

Conclusion

Abstract regular model checking is a useful technique for verifying infite-state and parameterized systems, accelerating the computation of reachable states (while also significantly reducing the state space explosion problem) by means of abstraction. The aim of this thesis was to create a tool for abstract regular model checking, which had so far only been implemented in prototypes.

The tool is written in C# using a Microsoft automata library as its back-end, and is suitable for both library and command-line use. Simple symbolic automata and transducers (whose definition is inspired by the FSA library [20]) are used to represent the systems being verified, and a number of different text formats are supported for loading and storing them (with the additional option of image generation). A configuration file with an easy-to-understand format is used for adjusting the many possible settings. The tool may also print out a clear description of its work (including generated automata) if so required. In addition to having a suitable interface, it is also written in a maintainable way and is fully documented.

The tool has been tested on a number of classic verification tasks. It has been used to verify the mutual exclusion property of several process synchronization algorithms, as well as the correct event ordering of a push-down system and a system with a queue. The tool also found a counterexample to an incorrectly specified version of Szymański's algorithm published in a dissertation [19].

A possible future extension is to add *abstract regular tree model checking* (which would require some automata type generalization and extending the parsing and printing algorithms) [6]. Other future work may include speed optimizations. The source code is available under the MIT license at GitHub [7].

Bibliography

[1] Baier, C.; Katoen, J.-P.: *Principles of Model Checking (Representation and Mind Series)*. The MIT Press. 2008. ISBN 026202649X, 9780262026499.

[2] Bartlett, K. A.; Scantlebury, R. A.; Wilkinson, P. T.: A Note on Reliable Full-duplex Transmission over Half-duplex Links. *Communications of the ACM*. vol. 12, no. 5. 1969: pp. 260–261. ISSN 0001-0782.

[3] Bieliková, M.: *Library for Finite Automata and Transducers*. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology. 2017.

[4] Bieliková, M.: Symboliclib. https://github.com/Miskaaa/symboliclib. 2017.

[5] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; et al.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *SAS 2006*. 2006.

[6] Bouajjani, A.; Habermehl, P.; Rogalewicz, A.; et al.: Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*. vol. 14, no. 2. Apr 2012: pp. 167–191.

[7] Chalk, M.: ARMC. https://github.com/matejchalk/ARMC. 2018.

[8] D'Antoni, L.; Veanes, M.: The power of symbolic automata and transducers. In *CAV'17*. Springer. July 2017.

[9] Esparza, J.; Hansel, D.; Rossmanith, P.; et al.: Efficient Algorithms for Model Checking Pushdown Systems. In *Proceedings of the 12th International Conference on Computer Aided Verification*. 2000. ISBN 3-540-67770-4. pp. 232–247.

[10] Genet, T.; Viet Triem Tong, V.: Reachability Analysis of Term Rewriting Systems with Timbuk. In *Proceedings of the Artificial Intelligence on Logic for Programming*. LPAR '01. Springer-Verlag. 2001. ISBN 3-540-42957-3. pp. 695–706.

[11] Hopcroft, J.: An n log n algorithm for minimizing states in a finite automaton. Technical report. Stanford University, Computer Science Department. 1971.

[12] Koutsofios, E.; North, S. C.: Drawing graphs with dot. Technical Report 910904-59113-08TM. AT&T Bell Laboratories. Murray Hill, NJ. 1991.

[13] Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*. vol. 17, no. 8. 1974: pp. 453–455. ISSN 0001-0782.

[14] Lengál, O.: Automata Benchmarks.
https://github.com/ondrik/automata-benchmarks. 2015.

[15] Lengál, O.; Šimáček, J.; Vojnar, T.: libVATA – A C++ library for efficient manipulation with non-deterministic finite (tree) automata.
https://github.com/ondrik/libvata. 2012.

[16] Lynch, N. A.; Patt-Shamir, B.: Distributed algorithms. 1993. lecture notes for 6.852, fall 1992.

[17] Mohri, M.; Pereira, F.; Riley, M.: General-purpose finite-state machine software tools. AT&T Labs – Research. 1997.

[18] Møller, A.; Schwartzbach, M. I.: *Static Program Analysis*. Department of Computer Science, Aarhus University. 2017.

[19] Nilsson, M.: *Regular Model Checking*. Licentiate Thesis. Information Technology, Uppsala University, Sweden. 2000.

[20] van Noord, G.: Finite State Automata Utilities.
http://www.let.rug.nl/vannoord/Fsa/. 1994–2000.

[21] Pryor, J.: Program option parser. http://www.ndesk.org/Options. 2008.

[22] Robinson, A.; Voronkov, A. (editors): *Handbook of Automated Reasoning*. Elsevier Science B.V.. 2001.

[23] Szymański, B. K.: A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait. In *Proceedings of the 2nd International Conference on Supercomputing*. 1988. ISBN 0-89791-272-1. pp. 621–626.

[24] Veanes, M.: Automata. https://github.com/AutomataDotNet/Automata. 2015.

[25] Vojnar, T.: Regular Model Checking. Technical report. Brno University of Technology, Faculty of Information Technology. 2015.

[26] Říha, J.: Automata. https://github.com/jakubriha/automata. 2017.

[27] Říha, J.: *An Efficient Functional Library for Finite Automata*. Master's Thesis. Brno University of Technology, Faculty of Information Technology. 2017.