Preface

The 8th International Conference on Implementation and Application of Automata (CIAA 2003) was held at the University of California, Santa Barbara, USA on July 16–18, 2003.

This volume of Lecture Notes in Computer Science contains the papers that were presented at CIAA 2003, as well as the abstracts of the poster papers that were displayed during the conference. The volume also includes the abstracts of two invited lectures presented by Thomas Henzinger and Juhani Karhumaki.

The 24 regular papers were selected from submissions covering various topics in the theory, implementation, and application of automata and related structures. Each submitted paper was reviewed by at least three program committee members, with the assistance of referees. The authors of the papers presented here come from the following countries: Brazil, Canada, Czech Republic, Finland, France, Germany, Ireland, Israel, Japan, The Netherlands, Poland, South Africa, Spain, Taiwan, and USA.

I wish to thank all who have made this meeting possible: the authors for submitting papers, the program committee members and external referees (listed in the proceedings) for their excellent work, and our two invited speakers. Finally, I wish to express my sincere appreciation to the sponsors, local organizers, proceedings chair, and the editors of the Lecture Notes in Computer Science series and Springer-Verlag, in particular Ms. Erika Siebert-Cole, for their help in publishing this volume.

July 2003

Oscar H. Ibarra
Conference Organization

Program Committee Chair

Oscar H. Ibarra, University of California, Santa Barbara, USA

Program Committee Members

Tevfik Bultan, University of California, Santa Barbara, USA
Jean-Marc Champarnaud, Université de Rouen, France
Zhe Dang, Washington State University, Pullman, USA
Jozef Gruska, Masaryk University, Czech Republic
Tero Harju, Academy of Finland, Finland
Tom Henzinger, University of California, Berkeley, USA
Juraj Hromkovic, RWTH Aachen, Germany
Kazuo Iwama, Kyoto University, Japan
Tao Jiang, University of California, Riverside, USA
Juhan Karhumaki, University of Turku, Finland
Lauri Karttunen, Xerox Palo Alto Research Center, USA
Sam Kim, Kyungpook University, South Korea
Nils Klarlund, AT&T Labs Research, USA
Werner Kuich, Technische Universität Wien, Austria
Carlos Martín-Vide, Universitat Rovira i Virgili, Spain
Denis Maurel, Université de Tours, France
Mehryar Mohri, AT&T Labs Research, USA
Gheorghe Paun, Romanian Academy, Romania
Jean-Eric Pin, CNRS and Université Paris 7, France
Grzegorz Rozenberg, Leiden University, The Netherlands
Kai Salomaa, Queen’s University, Canada
Helmut Seidl, Universität Trier, Germany
Imre Simon, University of Sao Paulo, Brazil
Wolfgang Thomas, RWTH Aachen, Germany
Derick Wood, HKUST, Hong Kong
Hsu-Chun Yen, National Taiwan University, Taiwan
Sheng Yu, University of Western Ontario, Canada

Organizing Committee Chair

Tevfik Bultan, University of California, Santa Barbera, USA
Organizing Committee Members

Constantinos Bartzis, University of California, Santa Barbara, USA
Zhe Dang, Washington State University, Pullman, USA
Omer Egecioglu, University of California, Santa Barbara, USA
Oscar H. Ibarra, University of California, Santa Barbara, USA
Hua Lee, University of California, Santa Barbara, USA
Theodora Lenard, University of California, Santa Barbara, USA

Proceedings Committee Chair

Zhe Dang, Washington State University, Pullman, USA

Steering Committee

Jean-Marc Champarnaud, Université de Rouen, France
Oscar H. Ibarra, University of California, Santa Barbara, USA
Denis Maurel, Université de Tours, France
Derick Wood, Hong Kong University of Science and Technology, Hong Kong
Sheng Yu, University of Western Ontario, Canada

Sponsors

Computer Science Department, University of California, Santa Barbara
College of Engineering, University of California, Santa Barbara
Graduate Division, University of California, Santa Barbara
AT&T Labs Research
Expertcity, Inc.

Referees

Cyril Allauzen                Orna Kupferman
Constantinos Bartzis         Charlotte Miller
Hans-Joachim Boeckenhauer    Jean Neraud
Romuald Boné                 Alexander Okhotin
Dirk Bongartz                Andrei Paun
Béatrice Bouchou             I. Petre
Michael Domaratzki           Brian Roark
C. Henkel                    Sebastian Seibert
Maia Hoeberechts             Walter Unger
Marcin Jurdzinski            Pascal Weil
Takeshi Koshiba              Hiroaki Yamamoto
# Table of Contents

## Invited Lectures

Automata for Specifying Component Interfaces .......................... 1  
*Thomas A. Henzinger*

Automata on Words (Extended Abstract) ................................. 3  
*Juhani Karhumäki*

## Regular Papers

Symbolic Synthesis of Finite-State Controllers  
for Request-Response Specifications ................................. 11  
*Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas*

Timing Parameter Characterization of Real-Time Systems .............. 23  
*Farn Wang and Hsu-Chun Yen*

Constructing Büchi Automata from Linear Temporal Logic  
Using Simulation Relations for Alternating Büchi Automata ............ 35  
*Carsten Fritz*

From Regular Weighted Expressions to Finite Automata .................. 49  
*Jean-Marc Champarnaud, Éric Laugerotte, Faissal Ouardi,  
and Djelloul Ziadi*

Bideterministic Automata and Minimal Representations  
of Regular Languages .................................................. 61  
*Hellis Tamm and Esko Ukkonen*

Succinct Descriptions of Regular Languages with Binary ⊕-NFAs .......... 72  
*Lynette van Zijl*

An Efficient Pre-determinization Algorithm ............................. 83  
*Cyril Allauzen and Mehryar Mohri*

Introducing VAUCANSON ........................................... 96  
*Sylvain Lombardy, Raphaël Poss, Yann Régis-Gianis,  
and Jacques Sakarovitch*

WFSC – A New Weighted Finite State Compiler ......................... 108  
*André Kempe, Christof Baeijs, Tamás Gaál Franck Guingne,  
and Florent Nicart*

Ternary Directed Acyclic Word Graphs .............................. 120  
*Satoru Miyamoto, Shunsuke Inenaga, Masayuki Takeda,  
and Ayumi Shinohara*
Running Time Complexity of Printing an Acyclic Automaton ............ 131
Franck Guingne, André Kempe, and Florent Nicart

Reducing the Time Complexity of Testing
for Local Threshold Testability ............................................. 141
Avraham Trakhtman

Branching Automata with Costs –
A Way of Reflecting Parallelism in Costs .................................. 150
Dietrich Kuske and Ingmar Meinecke

New Complexity Results for Some Linear Counting Problems
Using Minimal Solutions to Linear Diophantine Equations
(Extended Abstract) ...................................................... 163
Gaoyan Xie, Cheng Li, and Zhe Dang

TCTL Inevitability Analysis of Dense-Time Systems ...................... 176
Farn Wang, Geng-Dian Hwang, and Fang Yu

Conversation Protocols: A Formalism for Specification and Verification
of Reactive Electronic Services ........................................... 188
Xiang Fu, Tevfik Bultan, and Jianwen Su

Boolean Operations for Attribute-Element Constraints
(Extended Abstract) ...................................................... 201
Haruo Hosoya and Makoto Murata

XML Schema Containment Checking Based
on Semi-implicit Techniques ............................................. 213
Akihiko Tozawa and Masami Hagiya

Weak Minimization of DFA – An Algorithm and Applications ........... 226
Bala Ravikumar

Bag Automata and Stochastic Retrieval of Biomolecules in Solution ..... 239
Mark Daley, Mark Eramian, and Ian McQuillan

An Optimal Algorithm for Maximum-Sum Segment
and Its Application in Bioinformatics Extended Abstract .......... 251
Tsai-Hung Fan, Shufen Lee, Hsueh-I Lu, Tsung-Shan Tsou,
Tsai-Cheng Wang, and Adam Yao

Robust Parsing Using Dynamic Programming ............................. 258
Manuel Vilares, Victor M. Darriba, Jesús Vilares,
and Leandro Rodríguez

LR Parsing for Global Index Languages (GILs) ................................ 269
José M. Castaño

The Ehrenfeucht-Mycielski Sequence ..................................... 282
Klaus Sutner
Poster Papers

The Longest Common Subsequence Problem –
A Finite Automata Approach .............................................294
Bořivoj Melichar and Tomáš Polcar

AVA: An Applet for Visualizing FRACTRAN and Other Automata ......297
Nathan Preston and Mukkai S. Krishnamoorthy

Preliminary Experiments in Hardcoding Finite Automata ...............299
E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie

Computational Linguistic Motivations
for a Finite-State Machine Hierarchy ........................................301
Robert Kelly and Julie Carson-Berndsen

The Effect of Rewriting Regular Expressions
on Their Accepting Automata .............................................304
Michiel Frishert, Loek G. Cleophas, and Bruce W. Watson

Building Context-Sensitive Parsers from CF Grammars
with Regular Control Language ...........................................306
César Bravo and João José Neto

Finite-State Molecular Computing .........................................309
Olgierd Unold, Maciej Troc, Tadeusz Dobosz, and Alicja Trusewicz

Author Index ...........................................................311
Automata for Specifying Component Interfaces

Thomas A. Henzinger
University of California, Berkeley

Interfaces play a central role in component-based design and verification. An interface should specify no more and no less than the information necessary for checking if the corresponding component fits together with other components. This makes an interface specification different from a component specification. While component specifications are pessimistic (they must be satisfied in every environment), interface specifications are optimistic (they must be satisfiable by some environment). For example, assuming inputs \( x \) and \( y \) and output \( z \), the formula \( y \neq 0 \Rightarrow z = x/y \) is a component specification; the type declaration \( x: \text{int}; y: \text{int \setminus \{0\}}; z: \text{rat}; \) is an interface specification. While the former has no control over the input \( y \), the latter puts a design constraint on \( y \), namely, that \( y \) be different from 0. The division component can be composed with a client component if and only if the constraint \( y \neq 0 \) is guaranteed by the interface of the client. Note that not every input \( y \) satisfies the constraint \( y \neq 0 \), but it is a meaningful input constraint because it is satisfiable.

While for component specifications \( C \) and \( C' \), composition \( C || C' \) is a (more or less) total function, for interface specifications \( I \) and \( I' \), composition \( I || I' \) is partial: the composition is defined iff \( I \) and \( I' \) are compatible, that is, if the input constraints of \( I \) are guaranteed by the output constraints of \( I' \), and vice versa. For simple interfaces, which constrain input and output values, compatibility checking is type checking. We present a series of formalisms for specifying richer interfaces, which constrain input and output behavior and timing, as well as computational resources such as memory and power usage. For example, the interface of a file server with two methods \( A \) ("open file") and \( B \) ("read file") may require that \( B \) is not called before \( A \) is called. Such behavioral interfaces can be specified using input/output automata, and compatibility checking for automaton-based interfaces is a game-theoretic problem: the interface automata \( I \) and \( I' \) are compatible iff the environment has a way (i.e., strategy) to provide free inputs that prevent the composite automaton \( I || I' \) from entering incompatible states. For behavioral interfaces, an incompatible state arises when the input and output constraints of \( I \) and \( I' \) do not match; for example, if the server \( I \) expects a call of \( A \), but the client \( I' \) calls instead \( B \). For resource interfaces, an incompatible state arises when, by putting \( I \) and \( I' \) together, the available resources are exceeded. The requirement that the environment of \( I || I' \) follows a strategy that avoids incompatible states (provided such a strategy exists) is then the new input constraint of the composite interface.

An algebra of interfaces contains, in addition to parallel composition, also a preorder for refining (or abstracting) interfaces. While the refinement of component specifications treats inputs and outputs covariantly, as in trace containment or simulation, the refinement of interface specifications treats inputs
and outputs contravariantly, as in subtyping: a refined version of an interface may generate fewer outputs, but it must accept at least as many inputs. This leads to alternating trace containment and alternating simulation [1] for refining automaton-based interfaces.

This talk is based on joint work that originated in collaboration with Luca de Alfaro and is described in detail in the following series of papers: [6] formalizes the notions of compatibility and refinement between interfaces; [5] introduces input/output automata for specifying behavioral interfaces; [2] presents pushdown interfaces for software components, and [3] defines synchronous interfaces for hardware components; [7] enhances interfaces with real-time constraints, and [4] adds resource constraints to interfaces. All of these interface formalisms are variations on the common theme of defining interfaces as games between an input player and an output player.

References

Words, i.e. sequences of symbols, are basic notions in automata theory and, in fact, in any model of computing. In addition, properties of words are often fundamental tools when analyzing the behaviour of an automaton.

As examples we mention the following. Finite automata has played a crucial role in different variants of string matching and pattern matching problems, see [CR94]. The correctness of the algorithms are based, among other things, to the Periodicity Lemma of Fine and Wilf, which characterizes how long two periodic processes have to match in order to imply a common period. A simpler criterium used in these considerations is a characterization of the primitivity: a word $w$ is primitive if and only if it does not occur as a proper factor of its square $w^2$, see [CK97].

As another, maybe less well known example, we mention the following characterization of regular languages. Shirshov’s theorem, see [Lo83], is one of the jewels of combinatorics on words. It shows an unavoidable regularity of words, that is to say that something holds for all long enough words. One of the beautiful applications of Shirshov’s theorem is a combinatorial characterization of regular languages, see [RR83] and [dLV99]. A language is regular if and only if it is periodic (i.e. its syntactic monoid is periodic) and it possesses a so-called transposition property.

The list of the above examples can be made much longer. Actually, we are tempted to claim that whenever a fundamental property of words is revealed, it has applications on automata theory.

The relation between automata theory and combinatorics on words is, however, not at all one-sided. Results and techniques from automata theory can be used to prove properties of words. Sometimes the use of automata is mainly to clarify the presentation. However, occasionally properties of automata are essential to obtain results on words. Connections in this direction are the topic of this paper.

Connections of automata theory and other areas are not at all rare. On the contrary the very success of automata theory at its beginning was influenced by its connections to practical problems in computer science. In one hand, automata provided precise models for many properties of computers and programming languages. On the other hand, problems in computer science, like compiling, demanded to develop theoretical aspects of automata.

* Supported by the Academy of Finland under the grants 44087 and 202251.
Recently, it seems, that the applications of automata has become again much more important and diversified, see e.g. [K02]. Among these applications is that to combinatorics on words.

1 Terminology

We need only basic notions of finite automata and combinatorics on words, see [HU79] or [Be79] and [CK97] or [Lo02], respectively. The following lines are mainly to fix the terminology.

We denote by $A$ a finite alphabet. Words are finite or infinite sequences of elements of $A$. The corresponding sets are denoted by $A^*$ and $A^\omega$, respectively. The empty sequence is called the empty word and is denoted by $\epsilon$. Further $A^+ = A^* \setminus \{\epsilon\}$. The length of a word $u$ is denoted by $|u|$. A repetition of order $\rho \in \mathbb{Q}$, with $\rho \geq 1$, is a word $w = u^k u'$, where $u'$ is a prefix of $u$ and $\rho = |w|/|u|$. In particular, a square is a word of the form $uu$, where $u'$ is a prefix of $u$. An infinite word $w$ is called $\rho$-free (resp. $\rho^+$-free) if it does not contain as a factor repetitions of order larger than or equal to $\rho$ (resp. larger that $\rho$). An infinite word $w$ is ultimately periodic if it can be written in the form $w = uv\omega$ for some finite words $u$ and $v$. For a set $X \subseteq \Sigma^*$ we say that $w \in A^* \cup A^\omega$ has an $X$-factorization if $w$ can be expressed as a product of elements of $X$. Codes are those sets for which any word in $A^*$ has at most one $X$-factorization.

Some further notions are defined when needed.

We consider only finite automata, either without or with outputs. The latter are called finite transducers. The automata are used to compute also infinite words. We fix the terminology with an example.

Example 1. Consider a finite transducer depicted as in Fig. 1. Using $\{1\}$ as the initial and final state the transducer computes all words that have two different ways to write it as products of elements from the set $X = \{ab, abba, ba, baab, c\}$ such that one product starts with $abba$ and the other with $ab$. The transducer can easily be extended to a transducer computing all words having two (or more) different $X$-factorizations. In this case the transducer comes larger and not only due to symmetry. Indeed, it contains also the transition shown in Fig. 2. The above becomes more illustrative when we denote $x_1 = ab$, $x_2 = abba$, $x_3 = ba$, $x_4 = baab$ and $x_5 = c$. The complete transducer looks like that of Fig. 3.

Any minimal path from the state $\{1\}$ into itself spells a double factorization of a word. On the other hand, any word having a double factorization such that none of its prefixes nor suffixes has this property is spelt as a minimal path of the above type.

Consequently, the set of all double $X$-factorizations are computed by a finite transducer. In this transducer, as well as in Fig. 3, the names of the states lose their importance – although they are very illustrative in the first construction. The first component tells how much and the second which of the factorizations is ahead.
Let $Y = \{a, ab, bcc, ba, cccc\}$ and set $y_1 = a$, $y_2 = ab$, $y_3 = bcc$, $y_4 = ba$ and $y_5 = cccc$. Now, the transducer corresponding to the right part of Fig. 3 is shown in Fig. 4.

As we said transducers of Fig. 3 and 4 can be viewed as automata computing all double $X$-factorizations. Dually, these can be seen as devices defining infinite relations or equations such that $X$ and $Y$ are their solutions. This latter view is crucial in considerations of the next section. Note that the transducers defined above can be viewed also as devices computing infinite relations. Then no final states are used.

In order to summarize the above let $X = \{u_1, \ldots, u_n\}$ be a finite set of words and $\Xi = \{x_1, \ldots, x_n\}$ a set of unknowns of the same cardinality. Let $R(X) \subseteq \Xi^* \times \Xi^*$ be the set of relations satisfied by $X$ as described in the above example. Then our above considerations extend to

**Theorem 1.** The set $R(X)$ satisfied by a finite set $X \subseteq A^+$ is rational, that is computed by a finite transducer.

Obviously, the finiteness of $X$ is crucial in the above theorem.

## 2 Application 1

Let $X \subseteq A^+$ be a finite set of words. Then semigroup $X^+$ is a finitely generated subsemigroup of $A^+$. Such semigroups are called $F$-semigroups. Now a natural question is to ask whether two $F$-semigroups are isomorphic. In other words, whether they satisfy (under a suitable bijection of generators) exactly the same relations. Of course, the isomorphism requires that the generating sets are of the same cardinality. Similarly, one can ask whether an $F$-semigroup is embeddable into another $F$-semigroup. This is the case in our previous example: $Y^+ \hookrightarrow X^+$. Both of these questions are decidable:
Theorem 2. (CHK, 1997) It is decidable whether two given \( F \)-semigroups are isomorphic or one is embeddable into the other.

Proof. Let \( X = \{u_1, \ldots, u_n\} \) and \( Y = \{v_1, \ldots, v_n\} \) be two finite sets. We consider only the isomorphic case. Since, there exist only finitely many bijections from \( X \) into \( Y \), we can fix \( \Xi = \{x_1, \ldots, x_n\} \) and consider bijections \( u_i \mapsto x_i \) and \( v_i \mapsto x_i \) for \( i = 1, \ldots, n \). Note that necessarily the cardinalities of \( X \) and \( Y \) coincide.

Now, by Theorem 1, \( R(X) \) and \( R(Y) \) are rational relations. And we have to decide whether \( R(X) = R(Y) \). However, the equivalence problem for rational relations is undecidable!

To go around this the first observation is that, actually we are asking less than the equality of \( R(X) \) and \( R(Y) \), namely whether \( X \) is a solution of \( R(Y) \) and \( Y \) is a solution of \( R(X) \). But \( R(X) \) and \( R(Y) \) are infinite.

Now, a crucial point is that we consider these relations as systems of equations of the set \( \Xi \) of unknowns. Namely a fundamental property of word equation, so-called Ehrenfeucht Compactness Property states

**Lemma 1.** Each system \( R \) of equations over words and with a finite set of unknowns is equivalent with some of its finite subsystem \( R_0 \).

The proof of this beautiful result in general is based on Hilbert’s Bases Theorem, see Chapter 13 in [Lo02]. However, in the case of rational equations a direct proof is obtainable, and more importantly a finite subsystem \( R_0 \) can be found effectively. This is based on the pumping property of regular languages and a straightforward lemma on words, see [CHK97].

Now, we have all the ingredients to complete the proof. First we compute the relations \( R(X) \) and \( R(Y) \). Then we search for the corresponding finite equivalent subsets \( R_0(X) \) and \( R_0(Y) \). And finally we check whether \( X \) is a solution of \( R_0(Y) \) and \( Y \) of \( R_0(X) \). \( \square \)

The above proof is rather simple. However, natural extensions of the problem changes it drastically. An extension would be to consider regular sets instead of finite ones. Then, as a special case of our problem we could ask whether \( X \) is a code, or equivalently whether \( X^+ \) is a free semigroup. As is well known this problem is decidable, and essentially the same for finite and regular sets \( X \), cf. [BP85]. On the other hand, the isomorphism problem completely changes. Not only that our above proof would not work, but in fact we have an

**Open Problem.** Is the isomorphism problem for semigroups generated by regular sets decidable?

Another extension of our problem is obtained if instead of \( F \)-semigroups some other finitely generated semigroups are considered. Indeed, pretty soon the problem becomes undecidable. This is already the case for the multiplicative semigroups of \( 3 \times 3 \) triangular matrices over nonnegative integers, see [CHK99].
3 Application 2

Here we are looking for a borderline between a predictable and a chaotic behaviour on infinite words. More concretely, we try to point out the minimal number of a local periodicity which would imply the global periodicity. A local periodicity is defined as the requirement that prefixes of the word always end with a certain order of repetition, while the global periodicity means that the word is ultimately periodic. This research was initiated in a remarkable paper [MRS95], see also [MRS98], solving a conjecture of J. Shallit. Our presentation follows [KLP02] considering variants where automata theory comes particularly useful.

For the real number $\rho \geq 1$ and a natural number $p$ we say that a word $w$ is $(\rho, p)$-legal if it possesses as a suffix a repetition of order at least $\rho$ of a word of length at most $p$. So we could say that $w$ ends with a $(\rho, p)$-repetition.

Now, the question is which pairs $(\rho, p)$ imply the ultimate periodicity, i.e. when is the word ultimately periodic if all of its (long enough) suffixes are $(\rho, p)$-legal. We start with an example.

Example 2. Consider the famous infinite Fibonacci word

$$\alpha_F = abaababaabaab \ldots .$$

As is well known it is the fixed point of the morphism $F$: $a \mapsto ab$, $b \mapsto a$, in other words

$$\alpha_F = \lim_{i \to \infty} F^i(a).$$

Considering the ratio of a’s and b’s in prefixes of $\alpha_F$ one easily concludes that the word is not ultimately periodic. We claim that $\alpha_F$ is $(2, 5)$-periodic, that is any long enough prefix ends with a square of a word of length at most 5.

To prove the claim we first note, as is easy to see from the definition, that the $\{ab, aba\}$-factorization of $\alpha_F$ never contains two consecutive $ab$ and three consecutive $aba$ blocks. Hence suffix of $\alpha_F$ can be spelt from the graphs depicted in Fig. 5

Now, any word to the left and starting inside the rightmost block contains a short square at the beginning. For example, the upper line on the right graph starting from $b$ contains the square $abaababaab$.

\[
\begin{array}{c}
\cdots aba \quad ab \\
\downarrow \\
aba \quad ab \\
\downarrow \\
\cdots aba \\
\end{array}
\quad \begin{array}{c}
\cdots aba \quad ab \quad aba \\
\downarrow \quad \text{and} \quad \downarrow \\
aba \\
\end{array}
\]

\[
\begin{array}{c}
\cdots aba \\
\downarrow \\
\cdots aba \\
\end{array}
\quad \begin{array}{c}
\cdots aba \quad ab \\
\downarrow \\
\end{array}
\]

Fig. 5.
The above shows that the $(2,5)$-repititiveness does not imply the global periodicity. Amazingly, this is the optimal result as we now show.

**Example 3.** Here we compute all $(2,5)$-repetitive words. The idea is that we analyze how a word having a short square at the end can be extended by preserving the $(2,5)$-repititiveness. We start from the square $(abaab)^2$. The result is illustrated in the automaton $A_{2,5}$ of Fig. 6

A few things about this figure should be noted. First, we have omitted those extensions which lead only to ultimately periodic extensions. The $b$-extension from $(abaab)^2$ would be such one. Second, we have to remember not only the small squares at the end (which are required for the $(2,5)$-repititiveness) but also a bit more which might be needed in later steps. Third, all infinite words spelt from the automata are $(2,5)$-repetitive. Finally, since the above procedure from any other starting square would lead only to ultimately periodic words, the automaton $A_{2,5}$ characterizes all ultimately nonperiodic $(2,5)$-repetitive infinite binary words.

One interesting property of the automaton $A_{2,5}$ is that it contains intersecting loops. This immediately implies

**Theorem 3.** There exist nondenumerably many $(2,5)$-repetitive infinite words.

Let us continue the analysis of Example 3.

**Example 4.** Let us try to increase the order of repetition in $(2,5)$-repetitive words. So let us consider $(2^+,5)$-repetitive words. Of course, all $(2^+,5)$-repetitive words are $(2,5)$-repetitive as well. However, the node $(aba)^2(ab)^2$ is not any more $(2^+,5)$-repetitive. Indeed, independently of $X$ the word $Xabaabaabaabab$ does not contain a repetition of order larger than 2 of a word of length at most 5. Consequently, in the automata $A_{2,5}$ the intersecting loops break down. The conclusion is that all $(2^+,5)$-repetitive words are ultimately periodic.

We obtain a result showing that Theorem 3 is optimal:

**Theorem 4.**

(i) All $(2^+,5)$-repetitive words are ultimately periodic;

(ii) All $(2,4)$-repetitive words are ultimately periodic.
Part (i) was shown above, and Part (ii) can be shown by the arguments of Example 3.

As a conclusion, we have shown the exact amount of a local periodicity in terms of pairs \((\rho, p)\) which separates ultimately periodic, that is predictable words, from chaotic ones.

In [Le02] the similar very complicated analysis was done for all values of \(p\). If the length of the period is not bounded, then the above automata-theoretic approach would not work. This is the case which is solved in the seminal paper [MRS95]: the borderline value for the order of repetitions is \(1 + \varphi\), where \(\varphi\) is the number of golden ratio. In particular, no ultimately periodic word can always end with a cube, contrary to squares which was our starting point.

References


Symbolic Synthesis of Finite-State Controllers for Request-Response Specifications

Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas

Lehrstuhl für Informatik VII, RWTH Aachen
52056 Aachen, Germany
{wallmeier, phuetten, thomas}@i7.informatik.rwth-aachen.de

Abstract. We present a method to solve certain infinite games over finite state spaces and apply this for the automatic synthesis of finite-state controllers. A lift-controller problem serves as an example for which the implementation of our algorithm has been tested. The specifications consist of safety conditions and so-called “request-response-conditions” (which have the form “after visiting a state of \( P \) later a state of \( R \) is visited”). Many real-life problems can be modeled in this framework. We sketch the theoretical solution which synthesizes a finite-state controller for satisfiable specifications. The core of the implementation is a convenient input language (based on enriched Boolean logic) and a realization of the abstract algorithms with OBDD’s (ordered binary decision diagrams).

1 Introduction

Automatic verification (model-checking) is one of the most successful contributions of theoretical computer science to system development in industrial applications. A first central point of the model-checking methodology is provided by automata theoretic algorithms in connection with suitably chosen specification logics (usually temporal logics like CTL or LTL). A second prerequisite of efficient model-checking is the “symbolic approach”, namely the description of large state spaces and corresponding transition relations by Boolean formulas and their transformation into OBDD’s (ordered binary decision diagrams). Both techniques together have led to a powerful framework which supports large-scale system construction [1].

In the world of verification, the system to be implemented is assumed to be given and has to be “checked”. The automata theoretic methodology, however, provides theoretical algorithms for a much more ambitious task, namely for the automatic synthesis of control programs. Here a state-based system is modeled in a more structured way than by a simple transition graph: In order to capture the interactions between the control program and its environment, the global system is viewed as an arena for a two-person game, i.e. a transition graph where each state is associated to one of the two players (program = player 0, environment = player 1); and in a state associated to player \( i \) this player \( i \) is allowed to pick a transition to a successor state. Infinite system runs then correspond to...
infinite plays over this game graph (i.e. infinite paths through the graph); and the specification is completed by a winning condition on such plays which player 0 should try to fulfill. A solution of an infinite game over a finite arena requires to determine those states from which player 0 has a winning strategy, and in this case to construct such a strategy.

There is by now a large reservoir of algorithms and theoretical results on infinite games (see [6, 7, 4] for introductions). Only few algorithms have been implemented, however. For example, in [5] an implementation is presented for the algorithm [9] which solves parity games, motivated by the still open question whether parity games over finite graphs can be solved in polynomial time. Such an implementation is meant to support experiments in order to better understand an open theoretical question rather than to synthesize practical control programs. Synthesis tools for “real” control problems in the game-theoretic framework seem to be missing so far. This is due to three reasons:

1. lack of a powerful input language which allows to specify nontrivial game graphs (using Boolean formulas),
2. the conjunctive format of practical winning conditions,
3. the high complexity of the available synthesis algorithms.

Let us comment on the second and third aspect (which are connected). The classical automata theoretic winning conditions are the Büchi-, the Muller-, and the parity condition. The first simply requires that a certain set of states is visited again and again, the second and the third have the form of disjunctions where an atomic condition says “the set of infinitely often visited states is \( F \)”, respectively “the maximal color visited infinitely often is the (even) number \( i \)” (assuming that a coloring of states by numbers is given). Only the solution of games with a Büchi winning condition can (so far) be done in polynomial time. Moreover, a practical condition usually is a conjunction in which several simple conditions are collected. Its conversion into a disjunction (e.g. a Muller condition) and a subsequent construction of a winning strategy is too costly to allow nontrivial applications.

In the present paper, we describe a method and an implementation which provides progress regarding the first two items above and which gives some insight into the third. We proceed in three steps:

First, a specification language is provided for parameterized definitions of game graphs (using an enriched Boolean logic with Boolean quantifiers, indexed variables and support for binary encoding).

The winning conditions are supposed to be conjunctions of “safety conditions” and “request-response conditions”. Here a safety condition is represented by a restriction on the states to be visited or transitions to be taken. A request-response condition (sometimes also called “assume-guarantee-condition” in the literature) refers to two state-sets or state-properties \( P, R \) and has the form

\[ \text{"whenever a state in } P \text{ is visited, then now or later a state in } R \text{ is visited"} \]
It is a basic form of liveness condition. By an RR-game we mean a game where the winning condition is a conjunction of safety- and request-response-conditions. A large number of practical specifications for reactive systems can be coded in this format of RR-games.

Once the RR-game is specified, the second step consists in reducing it to a simple Büchi game (which will involve a problematic blow-up of the number of states). In a third step, the resulting Büchi game is solved. From this solution a finite-state controller for the original RR-game is extracted, respectively an error scenario is provided in terms of a strategy for the environment which falsifies controller’s winning condition.

We validate this approach by showing that one can solve small but concrete control problems which are coded as RR-games. As an example we provide an analysis and solution of a lift controller problem which is well below industrial dimensions but nevertheless too complicated to immediately suggest a common-sense solution. Not surprisingly, the complexity of implementing and evaluating conjunctions of request-response-conditions remains the main problem for the treatment of larger examples.

The paper is structured as follows: A short first section describes the example controller problem. The theoretical analysis is done in the subsequent two sections: first on the level of abstract states, secondly on the symbolic level (working with Boolean formulas). Then we give comments on our implementation, its application in the presented example, and some consequences to be drawn.

Our approach should be compared with recent work of Melcher [3] who emphasizes the modular construction of winning strategies. However, the winning conditions in [3] are only conjunctions of safety conditions and a single fairness condition of the form “if $P_1$ is visited infinitely often, then so is $P_2$”.

The task of handling long conjunctions of request-response conditions (and of fairness conditions) remains to be studied in more detail. The conclusion gives some hints on our current efforts in this context.

2 A Lift-Controller Problem

We consider a lift controller for two lifts in a building with $e$ floors. There are two players: the lift controller, which has to navigate both lifts, and the environment, represented by persons which request the lifts, respectively the target floors. It is convenient (but not essential) to assume that the two players act in alternating moves. Thus in a turn of the controller each of the two lifts is directed to one of the $e$ floors. In a turn of the environment the persons on the floors and in the two lifts can enter new requests. We distinguish between requests on the floors (that a lift should come) and requests in the lifts (to which floor to move). The first one can be fulfilled by either of the lifts whereas the second one only by the lift in which the request is issued.

Here are the requirements which the lift controller should to satisfy. This list of conditions is:
1. After a floor is requested a lift will eventually come to this floor. (If a request is issued inside a lift, this lift has to move to the destination.)
2. The highest and the ground floor are served directly ("express floors").
3. In the second floor is the post office: We assume a lift stopping there waits one extra move before proceeding (loading and unloading of mail).
4. The two lifts never stop at the same time in the second floor.
5. No lift skips a requested floor on its way.
6. A lift moves to an unrequested floor only if there is no open request.

The next two conditions are assumptions of the environment’s behaviour; they are added here to give the controller better chances to win:

7. At most one person enters a lift at a time (so that the new desired destination is unique).
8. At each moment, there are at least three floors which are not requested.

Some of the conditions above may seem artificial or very restrictive; for example, condition 2 is included in order to simplify the technical analysis for purpose of exposition. The results on satisfiability of the requirements would not be changed if condition 2 would just ask that only one of the "express floors" is to be served immediately.

We would like to answer the following question: For which values of $e$ can the lift controller fulfill all these requirements? Intuitively, increasing $e$ will make it harder to do this. For the cases where the requirements can be met, an appropriate control program (winning strategy of the corresponding infinite game) should be provided.

3 Theoretical Analysis over Abstract States

3.1 RR-Games

We apply the terminology of infinite two-person games as presented in [6], [4]. Player 0 acts as the controller and player 1 as the environment (the collection of people using the lift). The game is specified by a game graph and a winning condition for player 0. A game graph is of the form $G = (Q, E)$ where $Q = Q_0 \cup Q_1$ is the finite set of states ($Q_0$ the ones of the controller and $Q_1$ for the environment) and $E \subseteq Q \times Q$ is the transition relation. A sink state $s$ is always included allowing only a transition back to $s$. A play of a game is an infinite path in the game graph, i.e. an $\omega$-word $\rho = q_0 q_1 \ldots$ over the states with $(q_i, q_{i+1}) \in E$.

The winning condition specifies when player 0 wins a game $\rho$. (A play reaching the designated sink state $s$ and hence staying in this state is assumed to be won by player 1.) If player 0 can enforce to win when starting from state $q$ (so that the play will satisfy the winning condition), we say that player 0 wins from $q$. By $W_i$, we denote the winning region of player $i$, the set of states from which player $i$ wins. As usual, a strategy for player $i$ is a function which maps
any play prefix ending a state of \( Q_i \) to a state which can be visited next. We distinguish positional strategies and finite-state strategies. In the first case, the value only depends on the currently last state of a play prefix, in the second, the function is computable by a finite automaton over words from \( Q \) and with output alphabet \( Q \). The effective construction of finite-state strategies is thus a method for synthesizing control programs. For more details see e.g. [6].

We now code the example of Section 2 as a RR-game. The states represent the configurations of the system: Position of the lifts, currently requested floor numbers, and who’s turn it is to make the next move. Regarding the winning condition for player “controller”, we observe that the first requirement leads to a number of request-response-conditions. We distinguish which floor is requested and how the request arises: Note that this can happen by pressing a button on some floor (lift should fetch a person) or by pressing a button in one of the two lifts (choice of destination). This yields 3e individual request-response conditions from requirement 1. With a little reflection we can reduce this number to \( 3(e - 2) \), using the special role of the top and bottom floors (which have to be served directly). We can capture this requirement by cancelling all states and transitions from the game graph which violate this (observe that in this case the liveness condition reduces to a local condition).

All the remaining requirements are safety conditions. These conditions are encoded directly into the game graph by disallowing states and transitions which violate them. For example the last requirement that at least three floors are not requested means to cancel any state which does not satisfy this. The second requirement (ground floor and the highest floor are served directly) is a restriction on the transitions: The controller is allowed only to move the lift to one of these floors once they are requested, or else (when both are requested) moves to the sink state \( s \).

### 3.2 Reducing RR-Games to Büchi Games

An RR-game can be solved in two stages, first by a reduction to a Büchi game, which in a second step is solved by a well-known algorithm (using positional winning strategies). We follow the approach of game reductions as described in [8]. In the present case, this means to transform a RR-game over a graph \( G = (Q, E) \) into a Büchi game over a larger graph \( G' = (Q \times S, E') \), where the extra component \( S \) is used to simulate the RR-winning condition by a simple Büchi winning condition for a suitably defined set \( F \subseteq Q \times S \). (This amounts to a simulation of automata with RR-acceptance condition by Büchi automata.) Formally, we require that each play \( \rho = q_0q_1 \ldots \) over \( G \) induces a unique play \( \rho' = (q_0, s_0), (q_1, s_1), \ldots \) over \( G' \) such that player 0 wins \( \rho \) with the RR-condition iff (s)he wins \( \rho' \) with the Büchi condition. The construction will ensure the uniqueness of \( \rho' \) by a suitable initialization element \( s_0 \) and a deterministic updating of the components \( s_i \) given the sequence \( q_0q_1 \ldots \).

**Theorem 1.** RR-games are reducible to Büchi games, involving a blow-up from \( n \) states to \( nr2^{r+1} \) states if \( r \) request-response conditions are involved.
There is a family of RR-games for $P_1$ or $P_i$ is activated in state $z \in G$. The number of nodes of $i$ in state $z$ with $q' \in Q_i$ is in the winning region $W \times \{0, 1\}$. It can be easily shown that for this Büchi game the above-mentioned claims (of the definition of reduction) are satisfied.

The next result shows that the exponential blow-up in $r$ is unavoidable. We use a family of games where the number of vertices is $O(r)$. Theorem 2. There is a family of RR-games $G' = (Q', \Omega')$ with $G'' = (V', E')$ and the set $F$ of final states are defined as follows:

- $Q' := Q \times 2^{(1, \ldots, r)} \times \{1, \ldots, r\} \times \{0, 1\}$
- $((q, M, m, f), (q', M', m', f')) \in E' \Leftrightarrow$
  - $(q, q') \in E$
  - $M' = (M \cup \{i \mid q' \in P_i\}) \setminus \{i \mid q' \in R_i\}$
  - $m' = \left\{ \begin{array}{ll} m & \text{if } m \in M' \\ (m \mod r) + 1 & \text{otherwise} \end{array} \right.$
  - $f' = \left\{ \begin{array}{ll} 0 & \text{if } m = m' \\ 1 & \text{otherwise} \end{array} \right.$
- $F = Q \times 2^{(1, \ldots, r)} \times \{1, \ldots, r\} \times \{1\}$

For $1 \leq i \leq r$ there are three different kinds of request-response-conditions:

- $(P_i, Q_i)$: This condition is requested only in state $v_i$ and can only be fulfilled in the states $v_{i+j} = z_j$ or $e_j$ with $j \neq i$.
- $(P_i, Q_j)$: Analogous, with $v_i$ for $v_j$, $z_j$ for $z_i$.
- $(P_i, Q_i)$: In connection with $(P_i, Q_i)$, $(P_{i+j}, Q_{i+j})$ this condition requires that after a $P_i$-visit the succeeding visit to $w_{i+1}$ either vertex $z_i$ or $e_i$ is visited. If the play continues via $y_i$, the next pair $(P_{i+1}, Q_{i+1})$ is called.

$Q'_i$ is an abbreviation for the conjunction of $Q'_i$ with $j \neq i$ and $1 \leq j \leq r$. The node $v_0$ is in the winning region $W_0$ of player 0 because of the following observations. Here we call a predicate $P$ active if a $P$-vertex is visited but the corresponding $Q$-visit still not realized ("$P$ is not fulfilled").

1. By reaching state $w_{i+1}$ not both $P_i$ and $P_j$ are active for $1 \leq i \leq r$, because if $P_i$ is activated in state $v_i$, $P_j$ is fulfilled by $Q_j$ at the same time (and vice versa for $P_j$).
2. While the play does not reach a state $e_i$, at least one $P_j$ is active when reaching $w_1$: the fulfillment of a $P_i$ results in the request of $P_{(i \mod r)+1}$ if player 1 does not move to $e_i$.
3. Player 0 can only win the game if exactly one $P_i$ is active when reaching $w_1$. If another $P_j$ with $i \neq j$ is active, player 1 moves to $e_k$ when reaching $z_k$ whereby at least $P_i$ or $P_j$ can not be fulfilled.

From these observations one can conclude that every strategy automaton needs at least $2^r \cdot r$ states: $2^r$ for storing whether $P_i$ is requested (this implies that $P_i$ is not requested) or not, and $r$ for storing the presently active $P_i$.

### 3.3 Solving Büchi Games

Büchi games are solved here along the lines of [6]. We shortly recall the main points, starting with the (simpler) reachability games. In a reachability game, a set $F$ of states is designated, and the winning condition for player 0 requires that some state of $F$ should be visited during the play under consideration.

**Remark 3.** For reachability games the winning regions $W_i$ are decidable and positional winning strategies are computable (both in polynomial time).

Let us recall the proof: We compute inductively over $i$ the sets

$$Attr_0^i(F) := \{ q \in Q \mid \text{from } q \text{ player 0 can reach in } \leq i \text{ moves the set } F \}$$
The inductive construction proceeds as follows:

\[
\begin{align*}
\text{Attr}_0^0(F) &:= F \\
\text{Attr}_0^{i+1}(F) &:= \text{Attr}_0^i(F) \cup \\
&\quad \{ q \in Q_0 \mid \exists (q, r) \in E : r \in \text{Attr}_0^i(F) \} \cup \\
&\quad \{ q \in Q_1 \mid \forall (q, r) \in E : r \in \text{Attr}_0^i(F) \}
\end{align*}
\]

Since the sequence \( \text{Attr}_0^i(F) \) is increasing and \( Q \) is finite, there exists some \( k \) where this sequence becomes constant and is the union of all sets \( \text{Attr}_0^i(F) \). Let \( \text{Attr}_0(F) \) be \( \text{Attr}_0^k(F) \) for the first such \( k \). This set consists of the states from which player 0 wins the reachability game. For the (positional) winning strategy for player 0 it suffices to pick, for any \( Q_0 \)-state in a set \( \text{Attr}_0^{i+1}(F) \), some transition to a state in \( \text{Attr}_0^i(F) \) (which exists by construction).

For the solution of Büchi games, we have to determine the states from which player 0 can force infinitely many visits in the given set \( F \).

**Remark 4.** For Büchi games the winning regions \( W_i \) and positional winning strategies are computable (both in polynomial time).

We compute a set \( \text{Recur}_0(F) \) which contains the \( F \)-states from which player 0 can force infinitely many revisits of \( F \). As a preparation we define a modification of the attractor \( \text{Attr}_0^+(F) \) (with those states from which one revisit of \( F \) is possible in at least one move). We let \( \text{Attr}_0^+(F) = \bigcup_{i \geq 0} \mathcal{A}_i \) where

\[
\begin{align*}
\mathcal{A}_0 &= \emptyset \\
\mathcal{A}_1 &= \{ q \in Q_0 \mid \exists (q, r) \in E : r \in F \} \cup \{ q \in Q_1 \mid \forall (q, r) \in E : r \in F \} \\
\mathcal{A}_i &\geq 1 : \mathcal{A}_i^+ = \mathcal{A}_i \cup \{ q \in Q_0 \mid \exists (q, r) \in E : r \in \mathcal{A}_i \} \cup \\
&\quad \{ q \in Q_1 \mid \forall (q, r) \in E : r \in (A_i \cup F) \}
\end{align*}
\]

\( \text{Recur}_0(F) \) can be defined inductively:

\[
\begin{align*}
\text{Recur}_0^0(F) &:= F \\
\text{Recur}_0^{i+1}(F) &:= F \cap \text{Attr}_0^+(\text{Recur}_0^i(F)) \\
\text{Recur}_0(F) &:= \bigcap_{i \geq 0} \text{Recur}_0^i(F)
\end{align*}
\]

The winning region of player 0 is now \( W_0 := \text{Attr}_0(\text{Recur}_0(F)) \); it is again easy to obtain a corresponding positional winning strategy.

## 4 Solution on the Symbolic Level

A game graph over the symbolic state space is presented in the form \( G = (V, \varphi_0, \varphi_1, \tau) \) where \( V = \{v_0, \ldots, v_n\} \) is a finite set of Boolean variables, \( \varphi_i(v_0, \ldots, v_n) \) is a Boolean formula for the states of player \( i \), and
τ(v₀, ..., vₙ, v₀', ..., vₙ') is the transition formula. A concrete state is then an assignment of all variables of V. It can be interpreted as a bit-vector. For the transition formula we use a copy 𝑉 = {v₀', ..., vₙ'} of V. The transition formula is defined over the variables from V and V'. The variables of V describe the source of a transition and V' its target. We assume that φ₀ ∧ φ₁ = false must hold (every state is owned by only one player).

The computations for solving reachability and Büchi games can be easily transformed to the symbolic level. For a formula λ, set Attr₀(λ) := λ and

\[
Attr₀⁺(λ) := Attr₀(λ) ∨ (φ₀ ∧ (τ ∧ Attr₀(λ)|_{V → V'})) ∨ (φ₁ ∧ ¬(τ ∧ ¬Attr₀(λ)|_{V → V'}))
\]

where |_{V → V'} means that the variables are renamed from v to v' for each v ∈ V. The notation |_v indicates a restriction to V by existential quantification of the remaining variables (of V'). A similar transcripton is possible for the definition of Attr⁺(F) and Recur⁺(F) as defined in the previous section.

The reduction to Büchi games can be transformed to the symbolic state space with the same idea as on the abstract level. For a given RR-game by a game graph \(G = (V, φ₀, φ₁, τ)\) with |V| = n and set of pairs \((φᵢ, ψᵢ)\) for \(i = 1, ..., r\) an expanded game graph \(\overline{G} = (\overline{V}, \overline{φ₀}, \overline{φ₁}, \overline{τ})\) will be constructed:

- \(\overline{V}\) with \((n + r + \lceil \log r \rceil + 1)\) variables:
  - \(v₀, ..., v_{n−1}\) for states of V
  - \(vₙ, ..., v_{n+r−1}\) to encode the requests (\(v_{n+i−1}\) is true if the \(i\)-th condition is requested)
  - \(v_{n+r}, ..., v_{n+r+[\log r]}−1\) to encode the next request which should be fulfilled (binary encoding)
  - \(v_{n+r+[\log r]}\) flag, if it is a final state

- transition formula \(\overline{τ}\) (here \(\text{bin}(x, y, z)\) states that from component \(y\) onwards the next \(z\) bits are the binary encoding of \(x\)):

\[
\overline{τ} = τ \land (\bigwedge_{i=0}^{r−1} v'_{n+i} \iff ((v_{n+i} \land \neg ψᵢ|_{\overline{V} → \overline{V}}) \lor φᵢ|_{\overline{V} → \overline{V}}))
\]

\[
\land (\bigvee_{i=0}^{r−1} \text{bin}(i, n + r, [\log r]) \land ((v'_{n+i} \land \text{bin'}(i, n + r, [\log r]) \land v'_{n+i+2r})
\]

\[
\lor (\neg v'_{n+i} \land \text{bin'}(i + 1 \mod r, n + r, [\log r]) \land v'_{n+i+2r}))
\]

\[
\land (\bigvee_{i=0}^{r−1} \text{bin}(i, n + r, [\log r]) \land \alpha|_{\overline{V} → \overline{V}})
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{φᵢ}|_{\overline{V} → \overline{V}}
\]

\[
\land \neg \overline{φ₀} \land \neg \overline{φ₁}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{ψᵢ}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ'}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ'''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ''''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ'''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ''''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ'''''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ'''''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ''''''}|_{\overline{V} → \overline{V}}
\]

\[
\land \bigwedge_{i=0}^{r−1} \overline{τ''''''}|_{\overline{V} → \overline{V}}
\]
\[ \varphi_0 = \varphi_0 \land \alpha \quad \text{and} \quad \varphi_1 = \varphi_1 \land \alpha \]

- final state formula \( \lambda = v_{n+r} + \lceil \log r \rceil \)

The part (i) ensures that the flag that the \( i \)-th condition is requested, is only true at the target state of a transition if it was or is now requested and the target state is not a fulfilling assignment of \( \phi_i \). If the marked condition is satisfied, the end state is a final state and the marker is set to the next condition (iib). Otherwise it is not a final state and the marker does not change (iia). Part (iii) guarantees that only valid values for the marker are possible in a source state of a transition and (iv) is the same for the target state.

5 Back to the Lift Controller

The coding of the lift controller problem as introduced in Section 2 is done with \( 2 \lceil \log e \rceil + 3e + 2 \) Boolean variables if \( e \) floors are involved:

- \( 2 \lceil \log e \rceil \) variables for the current positions of the lifts (binary encoding)
- \( e \) variables for signaling the requests from the floors
- \( 2e \) variables for the requests from inside the lifts
- one variable to determine the player
- one variable to capture the waiting condition for the post office floor

Due to lack of space we do not present here syntactic details of the input language (in particular, the use of Boolean quantifiers) and on the structure of the user interface. The reader may obtain an impression by inspecting screen-shots accessible via www-i7.informatik.rwth-aachen.de/~wallmeier.

The following table summarizes the computation results on the lift controller example with small floor numbers \( e = 3, 4, 5 \). The computation was done on an Intel P3-M 1GHz with 256MB memory.

The table shows first the numbers of floors, of introduced Boolean variables, of RR-pairs, of states and the time to compute the BDD representation (using the package BuDDy of Jørn Lind-Nielsen [2]). The dramatic (exponential) increase of states for the corresponding Büchi game is clear from the next column. The entry “game-time” refers to the reduction to Büchi games and their solution. The last two columns show the sizes of the winning regions. For \( e = 3, 4 \) the controller wins (so here the specification is satisfiable) except from the special sink state. Already for 5 floors, however, controller is beaten (its winning region is empty). An analysis of environment’s winning strategy exhibits the problem.

<table>
<thead>
<tr>
<th>floors</th>
<th>vars</th>
<th>RR-pairs</th>
<th>states</th>
<th>BDD-time</th>
<th>Büchi-states</th>
<th>game-time</th>
<th>( W_0 )</th>
<th>( W_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>15</td>
<td>3</td>
<td>25</td>
<td>18.34s</td>
<td>1,200</td>
<td>24.27s</td>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>6</td>
<td>673</td>
<td>27.73s</td>
<td>516,864</td>
<td>40.39s</td>
<td>672</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>9</td>
<td>12,913</td>
<td>40.86s</td>
<td>119,006,208</td>
<td>699.34s</td>
<td>0</td>
<td>12,913</td>
</tr>
</tbody>
</table>
for controller: The environment forces one lift to the second floor (post office) without any other request. This lift is blocked in the next turn of the controller. The environment can now requests two floors, one of them an express floor, the other one on the way to it. Now the controller can not satisfy requests 2. and 5. simultaneously.

6 Conclusion

We have described the algorithmic solution of a class of infinite games and applied this for synthesis of finite state-controllers. The implementation offers an input language based on parameterized Boolean logic, and uses logical specifications which are given by conjunctions of RR-conditions. This is in contrast to the standard games of theoretical interest, where the winning conditions are usually given as disjunctions (rather than conjunctions). To our knowledge the implementation is the first program which allows to study the controller synthesis problem using the algorithmic theory of infinite games.

From our experiments (as in the case of the lift controller) it is clear that the number of RR-conditions is the main bottleneck. The same effect is to be expected if other conjunctive winning conditions (like Streett conditions) are used. How to dissolve the blow-up involved in the reduction of RR-games to Büchi games? The obvious approach would be to treat the RR-conditions separately. In the present example, it happens that the negative result for $e = 5$ could be obtained already by considering a subset of the request-response-conditions (namely, by forgetting the requests from inside the lifts and only taking into account the “outside requests”). This decomposition cannot, however, work in general: Simple examples show that the conjunction of request-response-conditions $c_1, c_2$ cannot be handled by solving the games for $c_1, c_2$ individually and proceeding to the intersection of the two computed winning regions for player 0. Even if the winning region is obtained as such an intersection, the winning strategy cannot, in general, be computed in this way but has to be generated globally. Ongoing work tries to clarify under which assumptions such conjunctions can be treated incrementally (one by one), and how to support the user to keep the number of these conditions small.

Acknowledgment

We thank a referee for his remark that a lower bound for RR-games should be given (now provided in Theorem 2).

References


Timing Parameter Characterization of Real-Time Systems

Farn Wang and Hsu-Chun Yen

Dept. of Electrical Engineering, National Taiwan University
Taipei, Taiwan 106, Republic of China
{farn,yen}@cc.ee.ntu.edu.tw

Abstract. We investigate the problem of characterizing the solution spaces for timed automata augmented by unknown timing parameters (called timing parameter automata (TPA)). The main contribution of this paper is that we identify three non-trivial subclasses of TPAs, namely, upper-bound, lower-bound and bipartite TPAs, and analyze how hard it is to characterize the solution space. As it turns out, we are able to give complexity bounds for the sizes of the minimal (resp., maximal) elements which completely characterize the upward-closed (resp., downward-closed) solution spaces of upper-bound (resp., lower-bound) TPAs. For bipartite TPAs, it is shown that their solution spaces are not semilinear in general. We also extend our analysis to TPAs equipped with counters without zero-test capabilities.

1 Introduction

Timed automata have been a popular model in the research of formal description and verification of real-time systems [1, 2]. In real-world applications, systems are usually described with unknown parameters to be analyzed. Here we use the term timing parameters to refer to those parameters which are compared with clocks in either timed automata [4] or parametric TCTL formulae [14, 15, 16, 17]. A timed automaton extended with unknown timing parameters is called a timing parameter automaton (TPA). A valuation of unknown parameters making the goal state reachable in a TPA is called a solution. In this paper, we are mainly concerned with the following problem:

- The reachability solution characterization (RSC) problem: Given a real-time system \( A \) and a reachability predicate \( \eta \), formulate a representation for the solution space of \( A \) with respect to \( \eta \).

In [4], it has been shown that the emptiness problem becomes undecidable when three or more clocks are compared with unknown parameters in TPAs. Knowing such a limitation, a line of subsequent research has been focused on the solution characterization problem for a number of restricted versions of TPAs. See, e.g., [3, 7, 14, 15, 16, 17]. The positive results obtained in the last few years have all been focused on unknown timing parameters in the specification.
of logic formulae. But in practice, it is more likely that engineers will use unknown parameters in the system behaviour descriptions. Moreover, engineers will be more interested at knowing the condition for solution parameters valuations than at knowing whether there exists a solution parameter valuation. In this work, we identify three subclasses of TPAs and investigate the complexity issue of their timing parameter characterization problems. The three subclasses are called upper-bound TPAs, lower-bound TPAs, and bipartite TPAs. Consider a TPA and w.l.o.g., we assume that only \( \leq \) and \(<\) are used in the predicates of the TPA. An upper-bound parameter \( \theta \) is one that only appears to the right of an inequality operator (e.g., \( x - y \leq \theta, x < \theta \)), whereas a lower-bound parameter \( \theta \) appears to the left of an inequality operator (e.g., \( \theta \leq x - y, \theta < x \)). Upper-bound (resp. lower-bound) TPAs are those whose unknown parameters are all upper-bound (resp. lower-bound) parameters. Bipartite TPAs refer to those for which every unknown parameter is either a lower-bound parameter or an upper-bound parameter, but not both. Bipartite TPAs were considered in a recent article [10] in which the emptiness problem (undecidable for parametric timed automata [4]) was shown to be decidable for such automata.

Intuitively, what makes upper-bound (resp. lower-bound) TPAs easier to analyze, in comparison with their general counterparts, lies in the fact that for each of such TPAs, the solution space is upward-closed (resp. downward-closed). (A set \( S \) over \( k \)-dimensional vectors of natural numbers, for some \( k \), is called upward-closed (resp., downward-closed) if \( \forall x \in S, y \geq x \implies y \in S \) (resp., \( \forall x \in S, y \leq x \implies y \in S \)). It is well known that an upward-closed set (resp., downward-closed set) is completely characterized by its minimal (resp., maximal) elements, which always form a finite set although the set might not be effectively computable in general. As we shall see later in this paper, we are able to give a complexity bound for the sizes of the minimal elements for a given upper-bound TPA. Our analysis is carried out in a way similar to a strategy proposed in [13] (by Valk and Jantzen), in which a sufficient and necessary condition was derived under which the set of minimal elements of an upward-closed set is guaranteed to be effectively computable. (Note, however, that [13] reveals no complexity bounds for the sizes of the minimal elements.) Taking advantage of certain properties offered by timed automata, we are able to refine Valk and Jantzen’s approach to yield complexity bounds for the sizes of the minimal elements for the upward-closed sets associated with upper-bound TPAs, allowing us to characterize their solution spaces. This in turn answers the RSC problem for upper-bound TPAs. To a certain extent, our result supplements the work of [10] (in which the emptiness problem was shown to be decidable for bipartite TPAs) by tackling a more general problem. We are also able to extend our analysis to the model of upper-bound timing parameter vector addition systems with states (TPVASSs), each of which can be viewed as a TPA equipped with counters without zero-test capabilities. Once the sizes of minimal elements become available, finding all such elements can be done by exhaustive search using the region graph technique, although it would clearly be interesting to develop smarter (and more efficient) algorithms. Some complexity results are also derived
for lower-bound TPAs. For bipartite TPAs, we are able to show that their solution spaces are not semilinear in general, in spite of the fact that the emptiness problem is decidable [10].

We feel that the method developed in this paper for analyzing upward-closed sets is interesting in its own right. Our technique refines the strategy of [13] (which deals with computing the minimal elements of upward-closed sets) in the following sense. Although the approach proposed in [13] is powerful for showing decidability for a variety of problems in a unified framework, the lack of information regarding the nature of the underlying system makes the calculation of the size of the associated upward-closed infeasible. Our study shows that if a key step in the algorithm of [13] meets certain conditions, then the sizes of the minimal elements can be deduced. It would be interesting to seek additional applications of our technique.

2 Models of Parametric Timed Systems

Let $Z$ ($N$ and $R^+$, resp.) be the set of all integers (nonnegative integers, and nonnegative reals, resp.), and $Z^k$ ($N^k$, resp.) be the set of $k$-dimensional vectors of integers (nonnegative integers, resp.). Let $\mathbf{0}$ be the zero vector. Let $v(i), 1 \leq i \leq k$, denote the $i$-th component of a $k$-dimensional vector $v$. Given two vectors $u$ and $v (\in N^k)$, $u \leq v$ if $\forall 1 \leq i \leq k, u(i) \leq v(i)$, and $u < v$ if $u \leq v$ and $u \neq v$. We define the norm of $v$, denoted by $\|v\|$, to be $\max\{|v(i)| \mid 1 \leq i \leq k\}$, i.e., the absolute value of the largest component in $v$. For a set of vectors $V = \{v_1, \ldots, v_m\}$, the norm of $V$ is defined to be $\max\{|v_i| \mid 1 \leq i \leq m\}$. In our subsequent discussion, we let $N_{\infty} = N \cup \{\infty\}$ ($\infty$ is a new element capturing the notion of something being ‘arbitrarily large’, and for every $t \in N, t < \infty$ holds). We also let $N^k_{\infty} = (N \cup \{\infty\})^k = \{(v_1, \ldots, v_k) \mid v_i \in (N \cup \{\infty\}), 1 \leq i \leq k\}$. For a $v \in N^k_{\infty}$, we also write $\|v\|$ to denote $\max\{|v(i)| \mid v(i) \neq \infty\}$, (i.e., the largest component in $v$ excluding $\infty$) if $v \neq (\infty, \ldots, \infty)$; $\|(\infty, \ldots, \infty)\| = 1$. Unless stated otherwise, we always assume that numbers are represented in binary, and the size of a number $t \in N$ is $[\log_2 t]$.

A set $U (\subseteq N^k)$ is called upward-closed if $\forall x \in U, \forall y, y \geq x \implies y \in U$. An element $x (\in N^k)$ is said to be minimal if there is no $y (\neq x) \in U$ such that $y < x$. We write $\min(U)$ to denote the set of minimal elements of $U$. For an element $v \in N^k_{\infty}$, let $\text{reg}(v) = \{w \in N^k \mid w \leq v\}$. A set $D (\subseteq N^k)$ is called downward-closed if $\forall x \in D, \forall y, 0 \leq y \leq x \implies y \in D$. An element $x (\in N^k_{\infty})$ is said to be maximal if there is no $y (\neq x) \in D$ such that $y > x$. We write $\max(D)$ to denote the set of maximal elements of $D$. For a given dimension, it is well known that every upward-closed (resp., downward-closed) set has a finite number of minimal (resp., maximal) elements. However, such finite sets may not be effectively computable in general. In an article [13] by Valk and Jantzen, the following result was proven which suggests a sufficient and necessary condition under which the set of minimal elements of an upward-closed set is effectively computable:
Theorem 1. ([13]) For each upward-closed set $U(\subseteq N^k)$, $\min(U)$ is effectively computable iff for every $v \in N^k$, the problem ‘reg$(v) \cap U \neq \emptyset$?’ is decidable.

It should be noted that in Theorem 1, the lack of information regarding the nature of the underlying system makes the calculation of the sizes of the associated minimal elements infeasible.

2.1 Timing Parameter Automata (TPA)

Given a set $P$ of basic propositions, a set $X$ of clocks, and a set $H$ of unknown parameters, a state predicate $\eta$ of $P$, $X$, and $H$ has the following syntax rules.

$$\eta ::= \text{false} \mid p \mid x \sim c \mid x \sim \theta \mid \eta_1 \vee \eta_2 \mid \neg \eta_1$$

where $p \in P$, $x, y \in X$, $c \in N$, $\theta \in H$, $\sim \in \{\leq, <, =, \geq, >\}$, and $\eta_1, \eta_2$ are state predicates. Notationally, we let $B(P, X, H)$ be the set of all state predicates on $P$, $X$, and $H$. Parentheses and traditional shorthands like $\Rightarrow, \land$ can also be used.

Definition 1. (Timing Parameter Automata, State and Interpretation): A timing parameter automaton (TPA) is a tuple $(Q, q_0, X, H, \mu, E, \tau, \pi)$, where $Q$ is a finite set of modes (operation modes, or control locations), $q_0 \in Q$ is the initial mode, $X$ is a set of clocks with readings in $R^+$, $H$ is a set of parameter variables with values in $N$, $\mu$ is a mapping from $Q$ such that for each $q \in Q$, $\mu(q) \in B(\emptyset, X, H)$ is the invariance condition true in $q$, $E \subseteq Q \times Q$ is the set of transitions, $\tau : E \mapsto B(\emptyset, X, H)$ is a mapping which defines the transition-triggering conditions, and $\pi : E \mapsto 2^X$ defines the set of clocks to be reset during each transition. A state of TPA $A = (Q, q_0, X, H, \mu, E, \tau, \pi)$ is a pair $(q, \nu)$ such that $q \in Q$ and $\nu$ is a mapping from $X$ to $R^+$ (i.e., $\nu$ represents the current clock readings). Let $U_A$ be the state set of $A$. An interpretation $\mathcal{I}$ for $H$ is a mapping from $N \cup H$ to $N$ such that for all $c \in N$, $\mathcal{I}(c) = c$.

Given a TPA $A$ and an interpretation $\mathcal{I}$, $A^{\mathcal{I}}$ is the timed automaton obtained from $A$ with all parameters interpreted according to $\mathcal{I}$. Given a predicate $\eta \in B(P, X, H)$ and an interpretation $\mathcal{I}$, $\eta^{\mathcal{I}}$ is the new predicate obtained from $\eta$ with all parameters interpreted according to $\mathcal{I}$.

Definition 2. (Satisfaction of State Predicate with Interpretation): A state $(q, \nu)$ satisfies state predicate $\eta \in B(Q, X, H)$ with interpretation $\mathcal{I}$, written as $(q, \nu) \models_{\mathcal{I}} \eta$, iff

- $(q, \nu) \not\models_{\mathcal{I}} \text{false}$;
- $(q, \nu) \models_{\mathcal{I}} q'$ iff $q = q'$
- $(q, \nu) \models_{\mathcal{I}} x \sim \theta$ iff $\nu(x) \sim \mathcal{I}(\theta)$ where $\theta \in N \cup H$;
- $(q, \nu) \models_{\mathcal{I}} \eta_1 \vee \eta_2$ iff $(q, \nu) \models_{\mathcal{I}} \eta_1$ or $(q, \nu) \models_{\mathcal{I}} \eta_2$; and
- $(q, \nu) \models_{\mathcal{I}} \neg \eta_1$ iff $(q, \nu) \not\models_{\mathcal{I}} \eta_1$.

If for all $\mathcal{I}$, we have $(q, \nu) \models_{\mathcal{I}} \eta$, then we may write $(q, \nu) \models \eta$. □
Definition 3. (Transitions): Given two states $(q, \nu), (q', \nu')$, there is a **mode transition** from $(q, \nu)$ to $(q', \nu')$ in $A$ with interpretation $I$, in symbols $(q, \nu) \rightarrow_I (q', \nu')$, iff $(q, q') \in E$, $(q, \nu) \models_I \mu(q) \land \tau(q, q')$, $(q', \nu') \models_I \mu(q')$, $\forall x \in \pi(q, q')(\nu'(x) = 0)$, and $\forall x \not\in \pi(q, q')(\nu'(x) = \nu(x))$. \hfill $\square$

For ease of expression, given a state $\nu$ and a $\delta \in \mathbb{R}^+$, we let $(q, \nu) + \delta = (q, \nu + \delta)$ be the state that agrees with $(q, \nu)$ in every aspect except for all $x \in X$, $\nu(x) + \delta = (\nu + \delta)(x)$.

**Definition 4.** ($(q, \nu)$-run of Interpreted TPA): An infinite computation of $A = (Q, q_0, X, H, \mu, E, \tau, \pi)$ starting at state $(q, \nu)$ with interpretation $I$ is called a $(q, \nu)$-run and is a sequence $((q_1, \nu_1, t_1), (q_2, \nu_2, t_2), \ldots)$ such that

- $q = q_1$ and $\nu = \nu_1$;
- $t_1 t_2 \ldots$ is a monotonically increasing divergent sequence such that for each $t \in \mathbb{R}^+$, there is an $i \in \mathbb{N}$ such that $t_i \geq t$ (meaning that the run is diverging);
- for each integer $i \geq 1$ and for each real $0 \leq \delta \leq t_{i+1} - t_i$, $(q_i, \nu_i) + \delta \models_I \mu(q_i)$ (meaning that the invariance condition $\mu(q_i)$ continuously holds throughout the time interval $[t_i, t_{i+1})$); and
- for each $i \geq 1$, $A$ goes from $(q_i, \nu_i)$ to $(q_{i+1}, \nu_{i+1})$ because of
  - a mode transition, i.e., $t_i = t_{i+1} \land (q_i, \nu_i) \rightarrow_I (q_{i+1}, \nu_{i+1})$; or
  - time passage, i.e., $t_i < t_{i+1} \land (q_i, \nu_i) + t_{i+1} - t_i = (q_{i+1}, \nu_{i+1})$. \hfill $\square$

### 2.2 The Reachability Solution Characterization Problem

Let $0$ be the mapping that maps everything to zero. The initial state of a TPA $A = (Q, q_0, X, H, \mu, E, \tau, \pi)$ is $(q_0, 0)$. Given a TPA $A = (Q, q_0, X, H, \mu, E, \tau, \pi)$, a goal state-predicate $\eta \in B(Q, X, H)$, and an interpretation $I$, we say $\eta$ is **reachable in $A$ with $I$, in symbols $A \sim_I \eta$, iff there exist a $(q_0, 0)$-run = $((q_1, \nu_1, t_1), (q_2, \nu_2, t_2), \ldots)$ in $A$, an $i \geq 1$, and a $\delta \in [0, t_{i+1} - t_i]$, s.t. $(q_i, \nu_i) + \delta \models_I \eta$. An interpretation $I$ that makes $A \sim_I \eta$ is called a solution for $A$ and $\eta$. The set of all solutions forms the so-called solution space. With respect to a given pair of $A$ and $\eta \in B(Q, X, H)$, the problem of finding a proper characterization for the solution space of $A$ with respect to $\eta$ arises naturally in many real-world applications. Such a problem is called the **Reachability Solution Characterization (RSC)** problem. Throughout the rest of this paper, we write $RSC(A, \eta)$ to denote the solution space of automaton $A$ with respect to predicate $\eta$.

### 2.3 Lower-Bound, Upper-Bound, and Bipartite TPAs

One of the major motivations in this work is to find practical classes of TPAs for which we can develop algorithms for its RSC problem. First we need the following concepts. A predicate $\eta \in B(P, X, H)$ is in **literal form** iff in $\eta$, negation symbols only appear in front of elements in $P$; there is no negative signs immediately before inequality literals; and only $\leq$ and $<$ are used in inequality atoms. Every predicate can be transformed to a literal form in linear time.
A TPA $A = (Q, q_0, X, H, \mu, E, \tau, \pi)$ is in called literal TPA iff $\mu(q)$ is in literal form for all $q \in Q$; and $\tau(q, q')$ is in literal form for all $q, q' \in Q$. Every TPA can also be transformed to a literal TPA in linear time. In a literal TPA, if an unknown parameter $\theta$ appears to the right of an inequality operator in a literal (e.g. $x - y \leq \theta$, $x - y < \theta$, $x \leq \theta$, $x < \theta$), then $\theta$ is called an upper-bound parameter. If it appears to the left of an inequality operator in a literal (e.g. $\theta \leq x - y$, $\theta < x - y$, $\theta \leq x$, $\theta < x$), then it is called a lower-bound parameter.

**Definition 5. (Bipartite, Lower-Bound, and Upper-Bound TPAs):**

A bipartite TPA $A = (Q, q_0, X, H, \mu, E, \tau, \pi)$ is a literal TPA such that its set $H$ of lower-bound parameters and set $\overline{H}$ of upper-bound parameters are disjoint, i.e. $H \cap \overline{H} = \emptyset$. If $H = \emptyset$, then $A$ is also called an upper-bound TPA. If $\overline{H} = \emptyset$, then $A$ is also called a lower-bound TPA. A predicate $\eta$ is an upper-bound (resp., lower-bound) predicate if all of its constituent parameters are upper-bound (resp., lower-bound) parameters.

There are two interpretations on bipartite TPA which serve importantly in defining the computability of the RSC problem. The first is the maximum interpretation $I^M$ with which $I^M(\emptyset) = 0$ for all $\emptyset \in H$, and $I^M(\emptyset) = \infty$ for all $\emptyset \in \overline{H}$. The second is the minimum interpretation $I^m$ with which $I^m(\emptyset) = \infty$ for all $\emptyset \in H$, and $I^m(\emptyset) = 0$ for all $\emptyset \in \overline{H}$. Note that maximum and minimum interpretations are not really interpretations as we defined in Definition 1 which does not map parameters to $\infty$. While interpreting $(q, \nu) \models_I \eta$ and $(q, \nu) \models_I \eta$, we shall assume $c \sim \infty = true$ and $\infty \sim c = false$, where $\sim \in \{<, \leq\}$.

We assume the basic knowledge of region graph constructions for timed automata presented in [1]. Suppose the biggest timing constant used in $A$ and $\eta$ is $C_{A, \eta}$. In [1], given a timed automaton $A$ (or a TPA with an interpretation), a region for a state $(q, \nu)$ is a triple $(q, \gamma, \phi)$ such that

- $\gamma$ records the integer parts of clock readings, at $(q, \nu)$, up to $C_{A, \eta}$; (when a clock reading is bigger than $C_{A, \eta}$, it is represented as $\infty$.)
- $\phi$ records the total ordering of the fractional parts of zero and clock readings at $(q, \nu)$.

A region graph for a TCTL model-checking problem instance is constructed with region set as the node set and (timed and discrete) transition relation from regions to regions as the arc set. TCTL model-checking problems can be answered with CTL model-checking algorithms on region graphs. A rough bound on the number of regions in a region graph for $A^I$ was computed as $A(|Q|, C_{A, \eta}, |X|) = |Q|((2 + C_{A, \eta})(|X| + 1))^{|X|}$. It is possible to get a tighter bound.

**Lemma 1.** $A \sim_{I} \eta$ iff there is a run, of less than $(A(|Q|, C_{A, \eta}, |X|) - 1)/2$ time units long, from the initial state $(q_0, 0)$ to a state satisfying $\eta$.

From now on, we shall let $\Gamma_{A, \eta} = (A(|Q|, C_{A, \eta}, |X|) - 1)/2$ for convenience.

In subsequent sections, we consider the problem of characterizing $RSC(A, \eta)$ when both TPA $A$ and goal predicate $\eta$ are upper-bound (or lower-bound). It is worthy of noting that in either case, the clock constraints in $\eta$ can be built
into TPA $A$. Consequently, the RSC problem can further be modified into one with only state reachability. Due to space limitations, most of the proofs of our subsequent results are omitted.

3 Computing Minimal Elements for Systems with Upward-Closed Solution Spaces

3.1 Upper-Bound TPAs

Now consider upper-bound TPAs with upper-bound goal predicates. By establishing an ordering on the elements of $H$ (i.e., $H = \{\theta_1, \ldots, \theta_k\}$, for some $k$), an interpretation can now be regarded as a $k$-dimensional vector in $N^k$. With a slight abuse of notation, for an interpretation $I$, we write $I(\theta)$ to denote $I(i)$, where $\theta = \theta_i$. Given an interpretation $I$ and a $\Delta \geq 0$ ($\Delta \in N^k$), we define $I + \Delta$ as the new interpretation such that for all $\theta \in H$, $(I + \Delta)(\theta) = I(\theta) + \Delta(\theta)$.

The following lemma shows that the solution space for each upper-bound TPA w.r.t. an upper-bound goal predicate is upward-closed.

Lemma 2. For any upper-bound TPA $A$ and upper-bound goal predicate $\eta$, if $A \Rightarrow I \eta$ is true, then $\forall \Delta \geq 0, (A \Rightarrow I + \Delta \eta)$. In words, the set of interpretations satisfying $\eta$ is upward-closed.

In view of the above lemma, each solution $I$ can actually be regarded a representative for a convex space of solutions, called funnel of $I$. Given an interpretation $I$, we use $\langle I \rangle$ to represent the funnel pointing at $I$, i.e.,

$$\langle I \rangle = \{I + \Delta \mid \Delta \geq 0\}$$

$I$ is called the point of funnel $\langle I \rangle$. (Note that $\langle I \rangle$ has a unique minimal element, namely, $I$.) A set of funnels $\langle I_1 \rangle, \ldots, \langle I_m \rangle$ is called mutually independent iff each funnel is not a subset of the unions of the others, that is, $\forall 1 \leq i \leq m (\langle I_i \rangle \not\subseteq \bigcup_{1 \leq j \leq m; i \neq j} \langle I_j \rangle)$, or equivalently

$$\forall 1 \leq i < j \leq m \exists \alpha \in H \exists \alpha' \in H (I_i(\alpha) < I_j(\alpha) \land I_i(\alpha') > I_j(\alpha'))$$

Given an upper-bound TPA and an $\eta$, Lemma 2 suggests that $RSC(A, \eta)$ is upward-closed. Using the basic theory of timed automata (see, e.g., [2]), the problem of, given an extended interpretation $I \in N^k$, deciding ‘$\text{reg}(I) \cap RSC(A, \eta) \neq \emptyset$’ is clearly decidable. This observation, in conjunction with Theorem 1, yields the computability of the set of minimal elements of $RSC(A, \eta)$, although it reveals no information regarding the size of $\text{min}(A, \eta)$. In the remainder of this section, we shall take advantage of certain properties of timed automata to derive complexity bounds for computing the minimal elements of $RSC(A, \eta)$. Our analysis involves the following two steps.

- We first show that the solution space for upper-bound TPAs and $\eta$ is a finite union of funnels; and
With an inductive scheme on the number of unknown upper-bound parameters, we derive a finite bound on the magnitudes of parameter values of point solutions of the funnels in the finite union.

The position of an existent solution is important in identifying the finite structure of the solution space. Let $\mathcal{I}$ be the interpretation that maps every $\theta \in H$ to $1 + a$. Lemma 1 implies that if the solution space is nonempty, then $\mathcal{I}(|Q|, C_{A, \eta}, |X|^{-1/2})$ is a solution.

Let $\mathcal{J}$ be a partial interpretation of the parameters in $H$, that is, $\mathcal{J}$ is undefined for some parameters in $H$. For convenience, we write $\mathcal{J}(\alpha) = \infty$, if $\alpha$ is undefined in $\mathcal{J}$. (By doing so, $\mathcal{J}$ becomes a vector in $N^k$, where $k = |H|$.) We conveniently use $(\mathcal{J})$ as the union of all $\mathcal{I}$ such that $\mathcal{I}(\alpha) = \mathcal{J}(\alpha)$ for every $\alpha$ defined in $\mathcal{J}$ (i.e., $\mathcal{J}(\alpha) \neq \infty$). Notice that $(\mathcal{J}) = \{\mathcal{I} \in N^k \mid \mathcal{I}(\alpha) = \mathcal{J}(\alpha), \forall \alpha \neq \infty\}$. For instance, $(\langle 5, \infty, 2, \infty \rangle) = \{(5, x, 2, y) \mid x, y \in N\}$. Thus we may also write $\mathcal{I} \subseteq (\mathcal{J})$ if $\mathcal{I}$ agrees with $\mathcal{J}$ on every $\alpha$ defined in $\mathcal{J}$.

Given a partial interpretation $\mathcal{J}$, in symbols, we let $\bar{H}_J$ be the set of variables in $H$ uninterpreted by $\mathcal{J}$, that is $\bar{H}_J = \{\alpha \mid \mathcal{J}(\alpha) \text{ is undefined}\}$. In $(\mathcal{J})$, there can be non-solution interpretations for $A$ and $\eta$. The following notation is for the characterization of those solution interpretations in $(\mathcal{J})$. Given a $\mathcal{J}$, we let $\Omega_{A, \eta}^J$ be the space of solutions $\mathcal{I}$ for $A \rightsquigarrow \mathcal{I} \eta$ with $\mathcal{I} \subseteq (\mathcal{J})$. If $\mathcal{J}$ happens to be a total interpretation, then (1) $\Omega_{A, \eta}^J = (\mathcal{J})$ in case $A \rightsquigarrow J \eta$; and (2) $\Omega_{A, \eta}^J = \emptyset$ otherwise. For convenience, given a partial interpretation $\mathcal{J}$ and $a \in N$, we let $\mathcal{J}[\alpha := a]$ be a new partial interpretation that agrees with $\mathcal{J}$ in every parameter except that $\mathcal{J}[\alpha := a](\alpha) = a$.

**Lemma 3.** If $\mathcal{J}$ is a partial interpretation $\mathcal{J}$ and $\mathcal{I} \in \Omega_{A, \eta}^J$ is a total interpretation, then

$$\Omega_{A, \eta}^J = (\mathcal{J}) \cup \bigcup_{\alpha \in \bar{H}_J \text{ and } 0 \leq \mathcal{I}(\alpha)} \Omega_{A, \eta}^{\mathcal{J}[\alpha := a]}$$

The importance of Lemma 3 is that once we can find a solution interpretation $\mathcal{I}$ in the solution space, the lemma suggests a way to inductively and compositionally construct the solution space by means of unions of funnels. But according to Lemma 1, we do know how to find this special interpretation $\mathcal{I}$ based on a given partial interpretation. As we shall see later, the ability to find such an $\mathcal{I}$ plays a critical role in deriving a complexity bound for the size of $\min(A, \eta)$. To emphasize its importance, the following definition is given to capture the idea stated above.

- **(Small witness property.)** For a given partial interpretation $\mathcal{J}$, a `small' total interpretation $\mathcal{I} \in \Omega_{A, \eta}^J$ can be computed effectively.

### 3.2 Complexity Analysis

Let $(\infty, \ldots, \infty)$ be the partial interpretation that is undefined on everything. Lemma 3, together with the fact that $\text{RSC}(A, \eta) = \Omega_{A, \eta}^{(\infty, \ldots, \infty)}$, suggests an algorithm for computing the constituent funnels of $\text{RSC}(A, \eta)$, provided that
\( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) is computable for every partial interpretation \( \mathcal{J} \). That is, if we view \( \Omega_{A, \eta}^{\mathcal{J}} \) as a procedure-call with parameters \( A, \eta, \) and \( \mathcal{J} \), then we can construct the solution space representation by invoking \( \Omega_{A, \eta}^{(\infty, \ldots, \infty)} \).

By examining all components of Formulus (1), we find that every component in (1) is with straightforwardly known complexity except \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \). It is obvious that if we can find bounds on the vector \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) for each \( \mathcal{J} \), then we can multiply and sum up all the component complexities to derive the complexity for the RSC problem. The major difficulty is to carefully account for all the component complexities so that bounds can be derived. Let \( A^{\mathcal{J}} \) be the new timed automaton obtained from \( A \) by substituting every defined \( \theta \) in \( \mathcal{J} \); and substituting every undefined \( \theta' \) in \( \mathcal{J} \) for \( \infty \). The bounds can be obtained by using Lemma 1. That is, we can construct the region graph for \( A^{\mathcal{J}} \) and \( \eta \) and use the length of the longest simple path in the graph to bound the vector components in \( \mathcal{I} \). In the same reasoning of Lemma 1, we know that there is an interpretation \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) making \( A \Rightarrow_{\mathcal{I}} \eta \) iff \( A^{\mathcal{J}} \Rightarrow_{\mathcal{I}} \eta \). According to [1], the size of region graph is bounded by

\[
2|Q| \cdot ((|X| + 1)^{|X|} \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|})
\]

In a region graph of this size, the time-span of the shortest path from one region to another can always be bounded by \( |Q| \cdot (|X| + 1)^{|X|} \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|} \). According to the same reasoning of Lemma 1, we can bound each component of the \( \mathcal{I} \) with

\[
|Q| \cdot (|X| + 1)^{|X|} \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|}.
\]

**Lemma 4.** For every partial interpretation \( \mathcal{J} \), there is an \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) such that for every \( \theta \in \mathcal{H}^{\mathcal{J}} \), \( \mathcal{I}(\theta) \leq |Q| \cdot (|X| + 1)^{|X|} \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|} \).

We want to construct the inductive definition of the magnitude of \( C_{A^{\mathcal{J}}, \eta} \). That is, we want to define \( C_{A^{\mathcal{J}}, \eta} \) based on those partial interpretations which define one less parameters than \( \mathcal{J} \) does. The following lemma unwinds Formulus (2) to for a bound on the complexity of \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) for each \( \mathcal{J} \). For convenience, we let \( |\mathcal{J}| \) be the number of parameters defined in \( \mathcal{J} \).

**Lemma 5.** In Formulus (1), for every partial interpretation \( \mathcal{J} \), there is an \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) such that for every \( \theta \in \mathcal{H}^{\mathcal{J}} \), \( \mathcal{I}(\theta) \) is \( O((|Q| \cdot (|X| + 1)^{|X|}) \sum_{0 \leq i \leq |\mathcal{J}|} |X|^i \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|^i}) \).

**Proof:** Base case: \( |\mathcal{J}| = 0 \). In this case, the bound is \( |Q| \cdot (|X| + 1)^{|X|} \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|} \), which is exactly the size bound of the region graph for \( A \) and \( \eta \). This case is true according to Lemma 1.

The inductive hypothesis is that there is an \( \mathcal{I} \in \Omega_{A, \eta}^{\mathcal{J}} \) such that for every \( \theta \in \mathcal{H}^{\mathcal{J}} \), \( \mathcal{I}(\theta) \) is \( O\left((|Q| \cdot (|X| + 1)^{|X|}) \sum_{0 \leq i \leq |\mathcal{J}|} |X|^i \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|^i}\right) \). This means that in the induction step, the biggest timing constant used in \( A^{\mathcal{J}[\theta=a]} \) is of the same complexity. Thus we deduce, according to Formulus (2), that the size bound of the region graph for \( A^{\mathcal{J}[\theta=a]} \) is

\[
O(2|Q| \cdot (|X| + 1)^{|X|} \cdot \left((|Q| \cdot (|X| + 1)^{|X|}) \sum_{0 \leq i \leq |\mathcal{J}|} |X|^i \cdot (C_{A^{\mathcal{J}}, \eta} + 1)^{|X|^i}\right)^{|X|})
\]
\[ = O(2^{|Q| \cdot (|X| + 1)^{|X|}} \cdot (|Q| \cdot ((|X| + 1)^{|X|}))^{\sum_{0 \leq i \leq |J|} |X|} \cdot (C_{A, \eta} + 1)^{|X|^{1+|J|}} \cdot |X|) \]

Since \( |J| + 1 = |J[\theta := a]| \) and the reachability does not need a path time greater than the number of regions, thus the complexity is proven.  \( \square \)

Using Lemmas 3 and 5, we have the following:

**Theorem 2.** Given an upper-bound TPA \( A \) and an upper-bound predicate \( \eta \), the size of \( ||\min(RSC(A, \eta))|| \) is bounded by \( |X|^{c \times |H|} \), where \( |X| \) and \( |H| \) denote the numbers of clocks and unknown parameters in \( A \), respectively, and \( c \) is a constant.

Once the sizes of minimal elements become available, finding all such elements can be done by exhaustive search using the region graph technique, although it would clearly be desirable to develop smarter (and more efficient) algorithms.

Our technique can also be applied to analyzing upper-bound timing parameter vector addition systems with states (TPVASSs), which can be thought of as TPAs equipped with counters without zero-test capabilities. The number of counters is called the dimension. The reader is referred to [9, 11] for more about VASSs. By applying the standard region graph technique of timed automata as well as the technique for solving the boundedness problem of VASSs ([11, 12]), we are able to obtain the following result.

**Theorem 3.** Given an upper-bound TPVASS \( A \) and an upper-bound predicate \( \eta \), the size of \( ||\min(RSC(A, \eta))|| \) is bounded by \( (2^{m \times \log m} \times |X|)^{c \times |H|} \), where \( m, |X| \) and \( |H| \) denote the dimension, the numbers of clocks and unknown parameters in \( A \), respectively, and \( c \) is a constant.

### 3.3 A Unified Strategy for Analyzing Upward-Closed Sets

In what follows, we propose a framework using which the sizes of the minimal elements in an upward closed set can be calculated. The idea is the following. In [13], the key in proving decidability lies in the ability of, given an arbitrary \( v \in N^k_\mathbb{N} \), testing whether \( '\text{reg}(v) \cap U \neq \emptyset' \). Now suppose in addition to the ability to test \( '\text{reg}(v) \cap U \neq \emptyset' \), we are also able to compute the size of a witnessing vector \( w \) in \( \text{reg}(v) \cap U \), if such a vector exists. That is, the small witness property holds for the system under consideration. In this case, the following result can be proven along a line similar to the proof of Theorem 1 presented in [13]. More precisely,

**Theorem 4.** For each upward-closed set \( U( \subseteq N^k_\mathbb{N} ) \), if given a \( v \in N^k_\mathbb{N} \), a witness \( w \) for \( '\text{reg}(v) \cap U \neq \emptyset' \) (if one exists) can be computed such that \( ||w|| \leq f(||v||) \), for some function \( f \), then \( ||\min(U)|| \leq (\underbrace{f \circ \cdots \circ f}_{k})(1) \).

It would be interesting to seek additional applications of the above result.
4 Computing Maximal Elements for Systems with Downward-Closed Solution Spaces

For TPAs with unknown lower-bound timing parameters, we still want to find characterization for the solution interpretation. In this case, there is one thing worth noting: The solution space for the unknowns is downward-closed. Geometrically, this means that the solution space is a union of “bottom-up” funnels. For convenience, we call such bottom-up funnels cones, and can be characterized by the maximal solutions of those cones. If we can find the upper-bounds for the maximal solutions, if any, of those cones, then we can shape the solution space in this case.

Lemma 6. Given a lower-bound TPA $A$ and a lower-bound predicate $\eta$, if $A \leadsto \theta \eta$ for some $I$ such that there is an $\theta$ with $I(\theta) > \Gamma_{A: \eta}$, then for all $I'$ such that $I'$ agrees with $I$ on all parameters except $I'(\theta) > I(\theta)$, $A \leadsto I'.$

Lemma 6 implies that to search for maximal solutions in cones, we only have to check the reachability of $A \leadsto I \eta$ with $I(\theta) \leq \Gamma_{A: \eta} + 1$ for all $\theta$. If $I$ is a solution, then the cone characterized by

$$\bigwedge_{I(\theta) \leq \Gamma_{A: \eta}} \theta \leq I(\theta)$$

which puts no restrictions on those parameters $\theta$ with $I(\theta) = \Gamma_{A: \eta} + 1$. Thus a simple way to formulate the algorithm for the RSC problem of lower-bound TPAs is with the following formula for the corresponding solution space.

$$\bigvee_{I : A \leadsto I \eta \land \forall \theta, 0 \leq I(\theta) \leq \Gamma_{A: \eta} + 1} \bigwedge_{I(\theta) \leq \Gamma_{A: \eta}} \theta \leq I(\theta)$$

(3)

Theorem 5. Given a lower-bound TPA $A$ and a lower-bound predicate $\eta$, the size of $|\text{max}(A, \eta)|$ is bounded by a polynomial in $|X|$.

For lower-bound TPVASSs, the argument used in the proof of Lemma 6 does not work, since a ‘loop’ in the region graph may not be repeatable due to the possibility of a loss in the counter value. So far we have seen that for restricted subclasses such as upper-bound and lower-bound TPAs, their solution spaces are upward-closed and downward-closed, respectively. This, together with a recent result of [10] showing the emptiness problem to be decidable for bipartite TPAs, leaves us to wonder whether the solution space of a bipartite TPA remains semilinear or not. Following a result in [4] that the solution spaces for general TPAs are not necessarily semilinear, it is reasonably easy to show:

Theorem 6. The solution spaces of bipartite TPAs are not semilinear in general.

5 Conclusion

We have studied in detail the sizes of the minimal (maximal, resp.) elements of upward-closed (downward-closed, resp.) solution spaces associated with upper-bound (lower-bound, resp.) TPAs. A line of future research for upper-bound
TPAs (and TPVASSs) is to explore the possibility of manipulating and characterizing the computations and the solution spaces in a symbolic fashion. One way to do this, perhaps, is to take a closer look at data structures designed explicitly for upward-closed sets, such as the so-called sharing trees of [6]. Finding how tight our complexity bounds for upper-bound and lower-bound TPAs are remains a question to be answered.

Acknowledgments

The authors thank the anonymous referees for their comments and suggestions, which greatly improved the correctness as well as the presentation of this paper. This research was supported in part by National Science Council grant NSC 91-2213-E-002-073.

References

Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata

Carsten Fritz
Institut für Informatik und Praktische Mathematik, CAU Kiel, Germany
fritz@ti.informatik.uni-kiel.de

Abstract. We present a new procedure for the translation of propositional linear-time temporal logic (LTL) formulas to equivalent nondeterministic Büchi automata. Our procedure is based on simulation relations for alternating Büchi automata. Whereas most of the procedures that have been described in the past compute simulation relations in the last step of the translation (after a nondeterministic Büchi automaton has already been constructed), our procedure computes simulation relations for alternating Büchi automata in an early stage and uses them in an on-the-fly fashion. This decreases the time and space consumption without sacrificing the potential of simulation relations.

We present experimental results that demonstrate the advantages of our approach: Our procedure is faster than TMP but produces, on the average, automata of about the same size; LTL2BA is faster than our procedure but produces larger automata.

1 Introduction

Propositional linear-time temporal logic (LTL for short) is a popular language for the specification of system properties. The standard way of model checking an LTL spec against a system is to translate the negation of the spec into an equivalent nondeterministic Büchi automaton (which incurs an exponential blow-up), build the product of this automaton with the system, and check this product for emptiness—it is empty if and only if the system satisfies the spec.

Obviously, the size of the Büchi automaton for the LTL formula is crucial for the efficiency of the above procedure. But minimizing Büchi automata is computationally difficult: Even testing universality for nondeterministic finite automata on finite strings is PSPACE-hard [GJ79]. This implies that approximating a minimum-size $\omega$-automaton (up to a constant factor) is impossible in polynomial time unless $P = \text{PSPACE}$.

In practice, various heuristics are in use for state-space reduction of the resulting automata. Standard techniques are simplifications of the input formula using a set of rewrite rules [MP92], and modifications in the transition structure of the resulting Büchi automaton (cf. [EH00]). Quotienting with respect to...
simulation or bisimulation equivalences is a sophisticated example for the latter [DHW91, ESW01, GBS02]. In general, a transition system simulates another system if, for every computation in the simulated system, there is a matching (w.r.t. an acceptance condition) computation in the simulating system. That is, a preorder on the states of the automaton is computed such that (usually) equivalent states can be merged.

Our algorithm which we present in this paper is also based on simulation relations, but our approach is different in that we compute a simulation relation before the exponential blow-up (see above) occurs.

The algorithm. In order to save time and space, we do not compute a simulation relation for the nondeterministic Büchi automaton. In our algorithm, a so-called delayed simulation relation [ESW01] is computed for an intermediate alternating Büchi automaton. The intermediate automaton can be interpreted as just another way of writing down the LTL formula—especially, the alternating automaton is only linear in the length of the formula. Consequently, the computation of the relation is fast in comparison to other simulation-based approaches (in the best case, exponentially faster).

In the other approaches, the crucial step is to actually compute a simulation quotient; in the procedure presented here, quotienting of the alternating automaton is not a necessary step of the construction. Instead, we use the simulation relation for on-the-fly simplifications in the computation of the result, thus again speeding up the process and saving memory resources. The price of this of course is that the resulting automaton may still contain simulation equivalent states, but our experiments indicate that this drawback is compensated by the advantage of using alternating automata in an intermediate stage.

Our construction proceeds in three main steps. First, the LTL formula is translated, in a very direct way, to an alternating Büchi automaton in which every state has either a universal or existential modality and in which we allow transitions labeled with the empty word “$\varepsilon$” (Section 3). We then compute the delayed simulation relation on the states of this automaton, using a variant of the game rules of [FW02b] that takes $\varepsilon$-transitions into account (Section 4). In the third step, we translate the alternating automaton to a nondeterministic (i.e., non-alternating) Büchi automaton, using the method of [MI84]. In this translation, we use the simulation relation for on-the-fly simplifications (Section 5). (In a fourth step in the implementation, we also do some a-posteriori simplifications not based on simulation relations.)

One advantage of our approach is that other techniques can easily be integrated into the algorithm. To give a clearer picture of the key aspects of our procedure, we did not integrate simplifications of the input formula into our algorithm (aside from computing the formula’s negation normal form), but formula rewriting can improve the result.

In Section 6, we report experimental comparisons of a prototypical implementation [FT] of our algorithm with the programs LTL2BA [Odd] and TMP [Etc], using a tool of Tauriainen and Heljanko [TH02]. Our experiments show that the automata produced by our implementation are, on the average, as good (i.e., as
small) as the automata of TMP (the automata of LTL2BA are larger). But with
the complexity of the formulas increasing, our program becomes substantially
faster than TMP, while LTL2BA is the fastest of the three programs.

Related work. The theoretical foundations of our algorithm are mainly taken
from [FW02b], but only a restricted framework (letters instead of propositions,
no ε-transitions) is considered there. The game rules of Arnold and Santoca-

lale [AS03] are quite similar to the rules we use here.

Translators from LTL to Büchi automata include an implementation as part
of Holzmann’s model checker Spin [Hol], Etessami’s "Temporal Massage Par-
lor" (TMP) [Ete], LTL2BA [Odd] of Gastin and Oddoux, and “Wring” [Blo]
by Somenzi and Bloem.

Both the Spin algorithm of Etessami and Holzmann [EH00] and TMP are
based on an algorithm of Gerth, Peled, Vardi, and Wolper [GPVW95]. The Spin
algorithm uses direct simulation while the latest version of TMP uses delayed
simulation quotienting [ESW01] for the simplification of the resulting automaton.
The algorithm of LTL2BA [GO01] mainly relies on a set of simple yet efficient
modifications of intermediate (alternating and generalized) Büchi automata. The
Wring algorithm [SB00] also uses direct simulation quotienting. Note that the
output of Wring is a generalized Büchi automaton with propositional labels
on the states, while the other programs (including our implementation) produce
transition-labeled Büchi automata, so a real comparison to Wring is not possible.

For simulation in general, the reader is referred to [Mil71, HK95]. Bisimula-
tion was introduced by Milner and Park [Mil80, Par81]. Henzinger, Kupferman,
and Rajamani [HKR97, HR00] introduce fair simulation and bisimulation rela-
tions and give efficient algorithms for computing them. Etessami, Schuller, and
Wilke [ESW01] improve on these results using a variant of an algorithm of Jur-
dziński [Jur00]. Gurumurthy, Bloem, and Somenzi [GBS02] study fair simulation
quotienting. Alur, Henzinger, Kupferman, and Vardi [AHKV98] study ordinary
simulation for alternating transition systems.

2 Basic Definitions

We fix a finite, non-empty set of propositions Σ with typical elements a, b, c, . . . .
LTL formulas over Σ are defined inductively by (1) tt and a are LTL formulas
for every a ∈ Σ, and (2) if φ and ψ are LTL formulas, then so are ¬φ, φ ∨ ψ,
Xφ and φUψ.
LTL formulas are interpreted over \( \omega \)-words, i.e., over infinite sequences of subsets of \( \Sigma \). For every \( \omega \)-word \( w : \omega \to 2^\Sigma \), we define the relation \( \models \) as follows.

\[
\begin{align*}
  w &\models \top, \\
  w &\models a \text{ iff } a \in w(0), \\
  w &\models \neg \phi \text{ iff } w \not\models \phi, \\
  w &\models \phi \lor \psi \text{ iff } w \models \phi \text{ or } w \models \psi, \\
  w &\models X\phi \text{ iff } w[i..] \models \phi, \\
  w &\models \phi \mathrel{U} \psi \text{ iff } \exists i (w[i..] \models \psi \land \forall j < i (w[j..] \models \phi)),
\end{align*}
\]

where \( w[i..] \) is defined by \( w[i..](n) = w(i+n) \) for every \( n < \omega \). As usual, we will allow derived logical operators like \( \land, \lor, \rightarrow \), and the temporal operators \( \mathsf{V}, \mathsf{F}, \mathsf{G} \) defined by \( \phi \mathsf{V} \psi = \neg (\neg \phi \mathrel{U} \neg \psi) \), \( \mathsf{F} \phi = \top \mathrel{U} \phi \) and \( \mathsf{G} \phi = \top \mathrel{V} \phi \). The language of an LTL formula \( \phi \) is

\[
L(\phi) = \{ w \in (2^\Sigma)^\omega \mid w \models \phi \}.
\]

Following [EH00], the transitions of our automata are labeled by so-called terms over the set of propositions. A term is the (possibly empty) conjunction of positive and negative propositions, i.e., the set of terms over \( \Sigma \) is

\[
\text{term}_\Sigma := \left\{ \bigwedge_{p \in M} p \land \bigwedge_{q \in N} \neg q \mid M, N \subseteq \Sigma \right\}.
\]

For every \( \Gamma \subseteq \Sigma \), let \( t(\Gamma) = \bigwedge_{p \in \Gamma} p \land \bigwedge_{q \in \Sigma - \Gamma} \neg q \). We define a preorder \( \preceq \) on \( \text{term}_\Sigma \) by setting \( t_0 \preceq t_1 \) iff \( t_0 \to t_1 \) is a tautology (for every \( t_0, t_1 \in \text{term}_\Sigma \)).

A nondeterministic B"uchi automaton (NBA for short) over \( \Sigma \) is a tuple

\[
A = (Q, \Sigma, q_I, \Delta, F)
\]

where \( Q \) is a finite set of states, \( q_I \in Q \) is an initial state, \( \Delta \subseteq Q \times \text{term}_\Sigma \times Q \) a transition relation, and \( F \subseteq Q \) a set of accepting states. Such an automaton \( A \) accepts a word \( w : \omega \to 2^\Sigma \) if and only if there is a sequence of states \( (q_i)_{i<\omega} \) of \( A \) such that \( q_0 = q_I \) and for every \( i < \omega \), there is a \( t_i \) such that \( (q_i, t_i, q_{i+1}) \in \Delta \) and \( t(w(i)) \preceq t_i \), and there are infinitely many \( i < \omega \) such that \( q_i \in F \). The **language** of \( A \) is

\[
L(A) = \{ w \in (2^\Sigma)^\omega \mid A \text{ accepts } w \}.
\]

For \( q \in Q \), we will write \( A(q) \) for the automaton \( A \) with new initial state \( q \), i.e., \( A(q) = (Q, \Sigma, q, \Delta, F) \).

3 From LTL to Alternating B"uchi Automata with \( \varepsilon \)-Transitions

In the first step of our algorithm, after a straightforward conversion to negation normal form, the LTL formula is translated to an alternating B"uchi automaton with term- and \( \varepsilon \)-transitions, or \( \varepsilon \)-ABA for short. An \( \varepsilon \)-ABA is similar to an NBA as defined in Section 2, but with the following changes.
There is a partition of the set of states $Q$ into the sets $E$ and $U$, called the existential and universal states respectively.

The transition relation $\Delta$ is a subset of $Q \times (\text{term}_\Sigma \cup \{\varepsilon\}) \times Q$, that is, we also allow transitions labeled with the empty word.

We say that an $\varepsilon$-ABA $A$ is legal if there is no non-empty sequence $(q_i)_{i<n}$ of states of $A$ such that $(q_i, \varepsilon, q_{i+1}) \in \Delta$ for every $i < n - 1$ and $q_0 = q_{n-1}$. Throughout this paper, we will only consider legal $\varepsilon$-ABA.

We define the language of a legal $\varepsilon$-ABA $A = (Q, \Sigma, q_I, \Delta, E, U, F)$ as follows, using a game-theoretic approach. Given an $\omega$-word $w$, the word game $G(A, w)$ is the Büchi game

$$(P, P_0, P_1, Z, F')$$

where $P = Q \times \omega \times \{0, 1\}$ is the set of positions, $P_0 = U \times \omega \times \{0, 1\}$ is the set of positions of Player 0 (called Pathfinder), $P_1 = E \times \omega \times \{0, 1\}$ is the set of positions of Player 1 (called Automaton), $p_I = (q_I, 0, 1)$ is the initial position, $Z = \{((s, i, j), (s', i, 0)) \mid (s, \varepsilon, s') \in \Delta\} \cup \{((s, i, j), (s', i+1, 1)) \mid \exists t' \in \text{term}_\Sigma: t(w(i)) \sqsubseteq t' \land (s, t', s') \in \Delta\}$ is the set of moves, and $F' = F \times \omega \times \{1\}$ is the set of accepting positions.

That is, the word game can be viewed as being played in rounds, and a round ends if one of the players chooses a transition labeled by a term, in which case the third component of a position switches from 0 to 1 (in a legal $\varepsilon$-ABA, every round is finite). The winner is determined by the sequence of the initial states of the rounds: The main difference from other definitions is that not all visited states are taken into account for acceptance. The language $L(A)$ of an $\varepsilon$-ABA $A$ is

$$L(A) = \{ w \in (2^E)^\omega \mid \text{Automaton wins } G(A, w) \}. \tag{12}$$

The translation from LTL to $\varepsilon$-ABA is straightforward using the well-known equivalences $\phi U \psi \equiv \psi \lor (\phi \land X(\phi U \psi))$ and, dually, $\phi V \psi \equiv \psi \land (\phi \lor X(\phi V \psi))$, see, e.g., [Var96]. That is, for an LTL formula $\phi_0$, we define the $\varepsilon$-ABA $A(\phi_0)$ inductively as follows.

1. The initial state is $\phi_0$.
2. If $\phi$ is a state of $A(\phi_0)$ then
   - if $\phi = \varepsilon$, $(tt, tt, tt) \in \Delta$ is a transition of $A(\phi_0)$,
   - if $\phi = a$ or $\phi = \neg a$ for $a \in \Sigma$, then $(\phi, \phi, tt) \in \Delta$ and $tt \in Q$,
   - if $\phi = \psi \lor \rho$ or $\phi = \psi \land \rho$, then $(\phi, \varepsilon, \psi), (\phi, \varepsilon, \rho) \in \Delta$ and $\psi, \rho \in Q$,
   - if $\phi = X \psi$, then $(\phi, tt, \psi) \in \Delta$ and $\psi \in Q$,
   - if $\phi = \psi U \rho$, then $(\phi, \varepsilon, \rho), (\phi, \varepsilon, \psi \land X \phi) \in \Delta$ and $\rho, \psi \land X \phi \in Q$,
   - if $\phi = \psi V \rho$, then $(\phi, \varepsilon, \rho), (\phi, \varepsilon, \psi \lor X \phi) \in \Delta$ and $\rho, \psi \lor X \phi \in Q$,
   - if $\phi = F \psi$ or $\phi = G \psi$, then $(\phi, tt, \phi), (\phi, \varepsilon, \psi) \in \Delta$ and $\psi \in Q$.

Formulas of the form $\psi \land \rho$, $\psi V \rho$ and $G \psi$ are universal states, all other formulas are existential states. The set of accepting states contains all states of the form $tt$, $\psi V \rho$, $G \psi$. We also add states of the form $\psi \lor \rho$ and $X(\psi V \rho)$ to the accepting states. This does not change the language of the automaton, but with this definition, $A(\phi_0)$ is a weak alternating Büchi automaton [MSS86]. This property allows us to faster solve the simulation game (see Section 4).
Proposition 1 (cf. [Var96]). For all LTL formulas $\phi$ over $\Sigma$, $L(\phi) = L(A(\phi))$.

In our implementation, we also apply some simple rules to eliminate some of the $\varepsilon$-transitions in the computed $\varepsilon$-ABA. We will not further discuss this here.

4 The Simulation Game for $\varepsilon$-ABA

Next, we construct a simulation relation on the states of legal $\varepsilon$-ABA. This relation will be our main tool for on-the-fly simplifications.

Following [FW02b], our definition of simulation relations for $\varepsilon$-ABA is based on a basic game for which different winning conditions can be defined.

Let $A = (Q, \Sigma, q_I, \Delta, E, U, F)$ be a legal $\varepsilon$-ABA and $p_0, q_0 \in Q$. The basic simulation game $G(p_0, q_0)$ is played in rounds by two players, called Spoiler and Duplicator, who move two pebbles (a red and a green pebble) on the transition graph of $A$. A round ends if a term-labeled transition has been chosen for both pebbles; before that, arbitrarily many $\varepsilon$-labeled transitions can be chosen. We say that a pebble is free if no term-labeled transition has been chosen for it in the current round; else it is locked. At the beginning of a round, let the red pebble be placed on $p$ and the green pebble on $q$. Then, the players play as follows.

1. Spoiler chooses a term $t \in \text{term}_\Sigma$.
2. The progression of the round depends on the modes of $p$ and $q$, and on the statuses of the pebbles (free or locked). Initially, both pebbles are free, and the round ends if both pebbles are locked. A player moves a free pebble on a state $s$ by choosing a transition $(s, x, s') \in \Delta$ such that $x = \varepsilon$ or $x \in \text{term}_\Sigma$ and $t \sqsubseteq x$. The round continues with the pebble on $s'$. If $x \in \text{term}_\Sigma$, the moved pebble becomes locked for the remainder of the round. The following rules determine who of the players has to move which pebble.
   - If $p \in E$ and $q \in U$ and both pebbles are free, Spoiler moves one of the pebbles (he can choose which one).
   - If $p \in E$ and the red pebble is free, but $q \in E$ or the green pebble is locked, Spoiler has to move the red pebble.
   - Conversely, if $p \in U$ or the red pebble is locked, but $q \in U$ and the green pebble is free, Spoiler has to move the green pebble.

If these cases do not apply, Duplicator has to move a free pebble in a symmetric fashion, as follows.

   - If $p \in U$ and $q \in E$ and both pebbles are free, Duplicator chooses one of the pebbles and moves it.
   - If $p \in U$ and the red pebble is free while the green pebble is locked, Duplicator moves the red pebble.
   - And if $q \in E$, the green pebble is free, and the red pebble is locked, Duplicator moves the green pebble.

If a player has to move a pebble but cannot (because there is no appropriate transition), he loses early. Or else the sequence $(p_i, q_i)_{i<\omega}$ of the initial positions of the rounds determines the winner (cf. the rules of the word game in Section 3).
We use the winning condition for delayed simulation, i.e., Duplicator wins if, for every $n < \omega$, if $p_n \in F$ then there is an $m \geq n$ such that $q_m \in F$. The simulation game with this winning condition is denoted $G^{dc}(p_0, q_0)$.

**Definition 1.** The relation $\leq_{dc} \subseteq Q \times Q$ is defined by

$$p \leq_{dc} q \iff \text{Duplicator wins } G^{dc}(p, q) .$$

(13)

The relation $\leq_{dc}$ is a preorder, i.e., it is reflexive and transitive. For every legal $\varepsilon$-ABA $A$ and for all states $p, q$ of $A$, if $p \leq_{dc} q$ then $L(A(p)) \subseteq L(A(q))$ (cf. [FW02b]).

As a preorder, $\leq_{dc}$ induces an equivalence relation $\equiv_{dc}$ defined by

$$p \equiv_{dc} q \iff p \leq_{dc} q \text{ and } q \leq_{dc} p .$$

(14)

The following observations are important for an efficient computation of the simulation relation and for its use in an on-the-fly fashion.

Note that $\varepsilon$-ABA constructed according to Sect. 3 have a special structure. No strongly connected component of such an $\varepsilon$-ABA contains two equivalent states:

**Proposition 2.** Let $\phi$ be an LTL formula, and let $C$ be a strongly connected component of the transition graph of $A(\phi)$. Let $q_0, q_1 \in C$. If $q_0 \equiv_{dc} q_1$ then $q_0 = q_1$.

We will use this property in Section 5 for our on-the-fly simplifications.

Since the $\varepsilon$-ABA that we construct for LTL formulas are weak (see Sect. 3), our delayed simulation game is in fact a reachability game in an AND/OR-graph and can thus be solved asymptotically as fast as a direct simulation game, see [FW02b, Theorem 9].

If Spoiler can choose an arbitrary term at the beginning of a round, the size of the game graph is exponential in $|\Sigma|$, but this is not necessary. To reduce the size of the game, in our implementation the set of terms $S$ that Spoiler can choose from depends on the current position $\phi$ of the red pebble, i.e., $S = S(\phi)$. For example, if the red pebble is on a state $X\psi$ then $S(X\psi) = \{tt\}$, and $S(\psi U \rho) = S(\psi) \cup S(\rho)$. Nevertheless, we cannot completely avoid an exponential blow-up here, because

$$S(\psi V \rho) = S(\rho) \cup \{\psi' \land \rho' \mid \psi' \in S(\psi), \rho' \in S(\rho)\} .$$

(15)

That is, for an LTL formula $\phi$ of length $n$ the $\varepsilon$-ABA $A(\phi)$ has $O(n)$ states and $O(n^2)$ transitions.

But the simulation game graph of $G^{dc}(\phi, \phi)$ has $2^{O(k \log n)}$ positions and moves, where $k$ is the maximal nesting depth of the operator $V$ (e.g., for $(a V b) V (c V (a \land (b V c)))$, $k$ is 3).
5 Computing the Nondeterministic Büchi Automaton with On-the-Fly Simplifications

We use an algorithm of Miyano and Hayashi [MH84] for translating an alternating to a nondeterministic Büchi automaton. While running this algorithm, we apply a set of simplification rules on-the-fly, i.e., after the construction of every single state of the NBA.

We first recall the classical construction of Miyano and Hayashi and then give some necessary definitions before presenting our algorithm.

The Miyano-Hayashi construction eliminates universal branchings via a modified power set construction. Given an ABA \( A = (Q, \Sigma, q_I, \Delta, E, U, F) \), a state of the resulting NBA \( A_{nba} = (Q', \Sigma, q'_I, \Delta', E', F') \) is of the form \((M, N)\) with \( N \subseteq M \subseteq Q \) and \( N \cap F = \emptyset \). Informally, if the computation of the NBA is in a state \((M, N)\), the computation of \( A \) is in all the states of \( M \) simultaneously, and \( A_{nba} \) will be in an accepting state as soon as the computation branches ending in the states of \( N \subseteq M \) reach accepting states (especially, \((M, N)\) is an accepting state if \( N = \emptyset \)). For more details on the Miyano-Hayashi construction, see [MH84].

It is not necessary to find a definition for quotients of \( \varepsilon \)-ABA, because every possible minimization effect of such a quotienting is taken into account by our modified Miyano-Hayashi construction.

Instead of merging equivalent states, we will choose a standard representative for every simulation equivalence class. This representative is chosen topologically maximal in the transition graph of \( A \), i.e., if \( q' \) is our representative for an equivalence class \([q]_{de}\) then no state in \([q]_{de} \setminus \{q'\}\) is reachable from \( q' \). By Proposition 2, there is such a representative. The strongly connected components of the transition graph of \( A \) and their topological sorting can be computed in time \( O(|Q| \cdot |\Delta|) \) (or \( O(|\phi|^{3}) \), for an LTL formula \( \phi \) and \( A = A(\phi) \)). Let \( r: Q \rightarrow Q \) be the function that maps every state to the representative of its class.

We define the preorders \( \leq_{de} \) and \( \leq_{weak} \) on \( Q' \) as follows. The relation \( \leq_{de} \subseteq Q' \times Q' \) is defined by

\[
(M_0, N_0) \leq_{de} (M_1, N_1) \text{ iff } \forall q \in M_1 \exists p \in M_0 : p \leq_{de} q \land (q \in N_1 \rightarrow p \in N_0)
\]  

The relation \( \leq_{weak} \subseteq Q' \times Q' \) is defined by

\[
(M_0, N_0) \leq_{weak} (M_1, N_1) \text{ iff } M_1 \subseteq M_0 \text{ and } N_1 \subseteq N_0
\]

Note that \( \leq_{weak} \) is a subset of \( \leq_{de} \), and \( \leq_{de} \) in turn is a subset of the full delayed simulation relation \( \leq_{de} \) on \( Q' \) (which we do not compute).

The translation algorithm from \( A \) to \( A_{nba} \)

1. The initial state \( q'_I \) of \( A_{nba} \) is \( \{r(q_I)\}, \{r(q_I)\} \) or, if \( r(q_I) \in F \), \( \{r(q_I)\}, \emptyset \).
   We add \( q'_I \) to the auxiliary set of new states, i.e., \( newStates := \{q'_I\} \).
2. Choose a state \((M, N) \in newStates \), remove it from \( newStates \) and add it to \( Q' \).
3. Compute the transitions starting at \((M, N)\) using the method of Miyano and Hayashi. Let \(T_{(M,N)} \subseteq \text{term}_\Sigma \times (2^Q \times 2^Q)\) be the set of these transitions.
4. Replace every transition \((t, (K, L)) \in T_{(M,N)}\) with \((t, (K', L'))\) where \(K' = \min^{de}(r(K)) \cup \min^{de}(r(L))\) and \(L' = \min^{de}(r(L)) - F\), where \(\min^{de}(R) = \{q \in R \mid \forall r \in R : r \leq^{de} q \rightarrow r \equiv^{de} q\}\) for every \(R \subseteq Q\). This is justified by [FW02a, Cor. 8].
5. Delete every transition \((t, (K, L))\) from \(T_{(M,N)}\) for which there is another transition \((t', (K', L')) \in T_{(M,N)}\) such that \(t \sqsubseteq t'\), and either \((K, L) \preceq^{\text{weak}} (K', L')\) or \((K, L) \preceq^{de} (K', L')\) and \(N = \emptyset\) or \(L' = \emptyset\) (see [FW02a, Subsect. 7.4]).
6. For all \((t, (K, L)) \in T_{(M,N)}\), add \(((M, N), t, (K, L))\) to \(\Delta'\), and add \((K, L)\) to \(\text{newStates}\) if \((K, L) \notin Q'\).
7. If \(\text{newStates} \neq \emptyset\), continue with step 2.
8. The set of accepting states is \(F' = \{(M, N) \in Q' \mid N = \emptyset\}\).

The correctness of the above algorithm follows from Proposition 1 together with [MH84, FW02a].

**Theorem 1.** For every LTL formula \(\phi\), \(L(\phi) = L(\text{A}_{nba}(\phi))\).

In our implementation, we also apply some standard simplifications to the nondeterministic Büchi automaton which are not based on simulation but mainly on the structure of its strongly connected components. We will not elaborate on these techniques here. See [GO01] for more details.

## 6 Experimental Results

Our prototypical implementation “LTL → NBA” of the algorithm outlined in Sections 3 to 5 is accessible via CGI, see [FT]. See Figure 1 for a graphical sample output of LTL → NBA using the dot program of the Graphviz package [ATT].

We have used the very helpful tool \(\text{lbtt}\) of Tauriainen and Heljanko [TH02] to compare our implementation with the programs “Temporal Massage Parlor” of Etessami [Ete] and LTL2BA of Gastin and Oddoux [Odd]. With LTL2BA, we also performed tests with formula rewriting disabled.

The tool \(\text{lbtt}\) generates random LTL formulas. The user can adjust the length of the formulas and the relative frequency of the temporal operators.

The computation times should be read as rough indicators since our prototype is written in Python, which is a rather slow interpreted language, i.e., we suppose that an equivalent C program would be at least 10 times faster. The program TMP is written in ML while LTL2BA is implemented in C.
Fig. 1. Graphical output of LTL → NBA for the formula $F((Fb \lor (a \lor Fb)))$

Test 1. 1000 formulas and their negations, of length 8 to 10, with at most 3 different propositions and an equal frequency of the operators $\lor$, $\land$, $F$, $G$, $U$, $V$.

<table>
<thead>
<tr>
<th>Test 1</th>
<th>LTL → NBA</th>
<th>TMP</th>
<th>LTL2BA</th>
<th>LTL2BA without f. r.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. no. of states</td>
<td>3.54</td>
<td>3.59</td>
<td>3.76</td>
<td>4.61</td>
</tr>
<tr>
<td>Avg. no. of transitions</td>
<td>6.96</td>
<td>6.65</td>
<td>7.75</td>
<td>11.03</td>
</tr>
<tr>
<td>Total time (sec)</td>
<td>209.1</td>
<td>70.2</td>
<td>12.7</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Test 2. 1000 formulas and their negations, now of length 10 to 14.

<table>
<thead>
<tr>
<th>Test 2</th>
<th>LTL → NBA</th>
<th>TMP</th>
<th>LTL2BA</th>
<th>LTL2BA without f. r.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. no. of states</td>
<td>4.37</td>
<td>4.71</td>
<td>4.93</td>
<td>6.58</td>
</tr>
<tr>
<td>Avg. no. of transitions</td>
<td>10.07</td>
<td>9.71</td>
<td>12.28</td>
<td>19.86</td>
</tr>
<tr>
<td>Total time (sec)</td>
<td>425.0</td>
<td>197.1</td>
<td>12.9</td>
<td>13.4</td>
</tr>
</tbody>
</table>
Test 3. Again 1000 formulas and their negations, now of length 11 to 15 and with an increased frequency of the operators \( \mathbf{U} \) and \( \mathbf{V} \).

<table>
<thead>
<tr>
<th>Test 3</th>
<th>LTL → NBA</th>
<th>TMP</th>
<th>LTL2BA</th>
<th>LTL2BA without f. r.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. no. of states</td>
<td>5.06</td>
<td>5.71</td>
<td>5.74</td>
<td>8.18</td>
</tr>
<tr>
<td>Avg. no. of transitions</td>
<td>12.59</td>
<td>13.83</td>
<td>15.98</td>
<td>28.86</td>
</tr>
<tr>
<td>Total time (sec)</td>
<td>575.0</td>
<td>14737.1</td>
<td>13.1</td>
<td>12.9</td>
</tr>
</tbody>
</table>

Test 4. 1000 formulas and their negations, of length 15, with the same frequency of the operators \( \mathbf{U} \) and \( \mathbf{V} \) as in Test 3.

<table>
<thead>
<tr>
<th>Test 4</th>
<th>LTL → NBA</th>
<th>TMP</th>
<th>LTL2BA</th>
<th>LTL2BA without f. r.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. no. of states</td>
<td>5.80</td>
<td>6.55</td>
<td>6.68</td>
<td>10.29</td>
</tr>
<tr>
<td>Avg. no. of transitions</td>
<td>16.05</td>
<td>16.86</td>
<td>20.63</td>
<td>42.93</td>
</tr>
<tr>
<td>Total time (sec)</td>
<td>986.6</td>
<td>3001.9</td>
<td>13.4</td>
<td>26.0</td>
</tr>
</tbody>
</table>

Our tests demonstrate that our implementation yields a competitive quality of results. With the complexity of the formulas increasing, our implementation becomes faster than TMP. LTL2BA is extremely fast, but the resulting automata are considerably larger. The differences in the performance of LTL2BA with and without formula rewriting may indicate the potential of adding this feature to our implementation, although the effect will probably be less dramatic since many rewrite rules are implicitly contained in our approach.

It may be surprising that our automata are often smaller than the automata computed by TMP although TMP computes the full delayed simulation quotient. We suppose this is because the algorithm of [GPVW95] used in TMP produces automata that are less suited for simulation-based minimizations than the automata we compute via alternating automata and the Miyano-Hayashi construction.

For detailed testing protocols, see [Fri].

7 Conclusion

We have presented a new algorithm for the translation of LTL formulas to non-deterministic Büchi automata which makes use of simulation relations for alternating Büchi automata. Our prototypical implementation demonstrates the power of these simulation relations as a tool for state-space reduction: The computed automata are small by comparison (although we do not use formula rewriting), and the average computation is fast in comparison to a simulation-based approach as used in TMP.

The simulation relations can be used on-the-fly and can easily be combined with other simplification techniques not based on simulation, like formula rewriting and SCC-based simplifications.

1 TMP spent 13997.3 sec (nearly 4 hrs.) on the formula \( \neg((\mathbf{X}b)\mathbf{U}((\mathbf{G}a)\mathbf{V}c)\mathbf{U}(\mathbf{G}\neg a))\mathbf{V}a \), resulting in an automaton of 115 states (LTL → NBA: 7.5 sec, 50 states; LTL2BA: 0.04 sec, 83 states).
Acknowledgments

Thanks to Björn Teegen who implemented LTL → NBA’s parser for the LTL formulas, the web interface, and the interface to the dot program.

References


[Fri] C. Fritz. Home page. \texttt{http://www1.ti.informatik.uni-kiel.de/~fritz/}. 45

[FT] C. Fritz and B. Teegen. LTL → BA. \texttt{http://www1.ti.informatik.uni-kiel.de/~teegen/ABA-Simulation/ltl.cgi}. 36, 43


From Regular Weighted Expressions
to Finite Automata

Jean-Marc Champarnaud, Éric Laugerotte, Faissal Ouardi, and Djelloul Ziadi

L.I.F.A.R., University of Rouen, France
{jmc,laugerot,fouardi,ziadi}@dir.univ-rouen.fr

Abstract. In this article we generalize the concepts of position automaton and ZPC structure to the regular K-expressions. We show that the ZPC structure can be built in linear time in the size of the expression and that the associated position automaton can be deduced from it in quadratic time.

1 Introduction

Weighted automata are state machines used in a lot of practical and theoretical applications such as computer algebra, non-linear control systems, image compression, speech recognition or text processing. Regular weighted expressions allow us to encode infinite data via a finite writing. In 1961, Schützenberger shows the famous equivalence between weighted automata and regular weighted expressions. Recent papers deal with the conversion from a regular weighted expression into a weighted automaton. The first one due to Caron and Flouret \cite{3} works on a subset of the set of regular expressions. It builds the position automaton \cite{6,10} in cubic time. The second one due to Lombardy and Sakarovitch \cite{9} computes an equation automaton \cite{1,5,13}. In this paper, we extend the notion of ZPC structure \cite{12,13} to regular K-expressions. In boolean case, the complexity of the construction of the ZPC structure is linear space and time w. r. t. the size of the expression and the complexity of the construction of the position automaton is quadratic \cite{4}. We show here that these complexity issues hold when regular K-expressions are considered.

In Section 2, we first introduce the notion of K-expression and we define the language of a K-expression. Next we present a brief description of formal series. Finally, we give the definition of regular K-expressions and automata with multiplicities. In Section 3, we introduce the notion of position automaton associated to a regular K-expression.

In Section 4, we generalize the ZPC structure \cite{12,13} to regular K-expressions. Next we show that, using this structure, the computation of the position automaton can be done in quadratic time in the size of the regular K-expression.

2 Preliminaries

Let A be a finite alphabet, and \((\mathbb{K}, \oplus, \otimes, 0, 1)\) be a semiring (commutative or not). The operator star \(\oplus\) can be partially defined, the scalar \(y^\oplus \in \mathbb{K}\) being
a solution (if there exists) of the equation \( y \otimes x \oplus \top = y \) (or \( x \otimes y \oplus \top = y \)) [7].

In the following definition, we introduce the notion of \( \mathbb{K} \)-expression:

**Definition 1** \( \mathbb{K} \)-expressions over an alphabet \( A \) are inductively defined as follows:
- \( a \in A \) and \( k \in \mathbb{K} \) are \( \mathbb{K} \)-expressions,
- if \( F \) and \( G \) are \( \mathbb{K} \)-expressions, then \( (F + G), (F \cdot G), \) and \( (F^*) \) are \( \mathbb{K} \)-expressions.

When there is non ambiguity \( (F \cdot G) \) will be denoted \((FG)\). Let \( E \) be a \( \mathbb{K} \)-expression. We will denote \( AE \) the alphabet of \( E \). The linearized version \( \overline{E} \) of \( E \) is the \( \mathbb{K} \)-expression deduced from \( E \) by ranking every letter occurrence with its position in \( E \). Subscripted letters are called positions. The size of \( E \), denoted \(|E|\) is the size of the syntactical tree of \( E \). For example, if \( E = (\frac{1}{2}a^* + \frac{1}{3}b^*)^*a^* \), \( AE = \{a, b\}, \overline{E} = (\frac{1}{2} \cdot a_1^* + \frac{1}{3} \cdot b_1^*)^* \cdot a_3^*, A_{\overline{E}} = \{a_1, b_2, a_3\} \) and \(|\overline{E}| = 13 \).

In order to introduce the language associated with a \( \mathbb{K} \)-expression, we define the null term of a \( \mathbb{K} \)-expression \( E \), denoted \( c(E) \).

**Proposition 1** Let \( E \) be a \( \mathbb{K} \)-expression. The null term \( c(E) \) of \( E \) is inductively defined as follows:

\[
\begin{align*}
    c(k) &= k \quad \text{for all } k \in \mathbb{K}, \\
    c(a) &= 0 \quad \text{for all } a \in A, \\
    c(F + G) &= c(F) + c(G), \\
    c(FG) &= c(F)c(G), \\
    c(F^*) &= c(F)^*.
\end{align*}
\]

The null term of \( E = (\frac{1}{2}a^* + \frac{1}{3}b^*)^*a^* \) is \( c(E) = (\frac{1}{2}0^* + \frac{1}{3}0^*)0^* \).

In the following, we denote by \( T_E \) the set of subexpressions of the \( \mathbb{K} \)-expression \( c(E) \). The next definition introduces the concept of language associated to a \( \mathbb{K} \)-expression.

**Definition 2** Let \( E \) be a \( \mathbb{K} \)-expression. The regular language \( L(E) \) over \((A \cup T_E)^*\) associated to \( E \) is inductively defined as follows:

\[
\begin{align*}
    L(k) &= \emptyset, \\
    L(a) &= \{1a1\}, \\
    L(F + G) &= L(F) \cup L(G), \\
    L(FG) &= L(F)L(G) \cup \{c(F)\}L(G) \cup L(F)\{c(G)\}, \\
    L(F^*) &= \{c(F)^*\}(\bigcup_{i \geq 1}(L(F)\{c(F)^*\}))^1.
\end{align*}
\]

For example the language associated to the \( \mathbb{K} \)-expression \( E = (\frac{1}{2}(a)^*) \) is \( L(E) = \{0^*1a10^*, \frac{1}{2}0^*1a10^*1a10^*, \cdots \} \). Note that according to the formulas of Definition 2, words in \( L(E) \) have the form \( \alpha_11a_1\alpha_21a_21\cdots \alpha_m1a_{m+1} \), where \( \alpha_i \in T_E \).
In the following, we present a brief description of formal series. Then we define a subset of the set of $K$-expressions, usually called regular $K$-expressions which are associated to regular series [2].

**Definition 3** A (non-commutative) formal series with coefficients in $K$ and variables in $A$ is a map from the free monoid $A^*$ to $K$ which associates with the word $w \in A^*$ a coefficient $(S, w) \in K$.

A formal series is usually written as an infinite sum: $S = \sum_{w \in A^*} (S, w) w$. The support of the formal series $S$ is the language $\text{supp}(S) = \{ w \in A^* \mid (S, w) \neq 0 \}$. The set of formal series over $A$ with coefficients in $K$ is denoted by $K \langle\langle A \rangle\rangle$.

A structure of semiring is defined on $K \langle\langle A \rangle\rangle$ as follows [2, 8]:

- $(S + T, w) = (S, w) \oplus (T, w)$,
- $(ST, w) = \bigoplus_{uv=w} (S, u) \otimes (T, v)$, with $S, T \in K \langle\langle A \rangle\rangle$.

A polynomial is a formal series with finite support. The set of polynomials is denoted by $K \langle A \rangle$. It is a subsemiring of $K \langle\langle A \rangle\rangle$. The star of series is defined by:

$S^* = \sum_{n \geq 0} S^n$ with $S^0 = \varepsilon$, $S^n = S^{n-1} S$ if $n > 0$. Notice that the star of a formal series does not always exist.

**Proposition 2** [8] The star of a formal series $S \in K \langle\langle A \rangle\rangle$ is defined if and only if $(S, \varepsilon)^K$ is defined in $K$.

**Definition 4** The semiring of regular series $K \text{rat}(A^*) \subset K \langle\langle A \rangle\rangle$ is the smallest set of $K \langle\langle A \rangle\rangle$ which contains the polynomials semiring $K \langle A \rangle$, and which is stable by the operations of addition, product and star when this latter is defined.

We define the mapping $ev$ which associates to each term $t \in T_E$ its evaluation in $K$.

**Definition 5** The mapping $ev$ is recursively defined from $T_E$ to $K$ as follows:

$ev(t) = T \in K$ if $|t| = 1$,

$ev(t_1 + t_2) = ev(t_1) \oplus ev(t_2)$,

$ev(t_1 t_2) = ev(t_1) \otimes ev(t_2)$,

$ev(t^*) = ev(t)^\oplus$.

The following definition introduces the notion of regular $K$-expression which allows us to represent regular series by finite writing.

**Definition 6** A regular $K$-expression is defined inductively by:

- $a \in A, k$ such as $T \in K$ are regular $K$-expressions which respectively denote the regular series $S_a = a$ and $S_k = T$,
- if $F, G$ and $H$ (s.t. $ev(c(H)^*)$ exists) are regular $K$-expressions which respectively denote the regular series $S_F$, $S_G$ and $S_H$, then $F + G, FG$, and $H^*$ are regular $K$-expressions respectively denote the regular series $S_F + S_G$, $S_F S_G$ and $S_H^*$. 

By Proposition 2, regular $K$-expressions are $K$-expressions where the evaluation of the null term is well-defined.

**Definition 7** Let $A$ be a finite alphabet, and $K$ be a semiring (commutative or not). We define an automaton with multiplicities $A = (Q, q_0, \delta, F, \mu)$ as follows:

- $Q$ is a finite set of states,
- $q_0$ is the initial state with 1 as initial coefficient,
- $\delta : Q \times (KA) \rightarrow 2^Q$ is the transition function,
- $F \subseteq Q$ is the set of final states,
- $\mu : Q \rightarrow K$ is the final function. We have $q \in F$ if and only if $\mu(q) \neq 0$.

The definition of an automaton with multiplicities is in fact more general but this one is sufficient for our construction.

A recognized path $c$ in $A$ is a sequence $(q_0, q_1), (q_1, q_2), \ldots, (q_{n-1}, q_n)$ of transitions in $A$. It is written $c = (q_0, q_1, q_2, \ldots, q_{n-1}, q_n)$. We denote by $\text{coef}(c)$ the cost of the path $c$ in $A$. Formally, $\text{coef}(c) = \alpha_{q_1} \otimes \alpha_{q_2} \otimes \cdots \otimes \alpha_{q_n} \otimes \mu(q_n)$ with $\delta(q_{i-1}, \alpha_i a_{q_i}) = q_i$, where $q_i \in Q$, $a_{q_i} \in A$ and $\alpha_{q_i} \in K$ for all $1 \leq i \leq n$. We denote by $m(c)$ the word obtained in $A$ by going across the path $c$, i.e., $m(c) = a_{q_1} a_{q_2} \cdots a_{q_n}$ where $a_{q_i} \in A$ for all $1 \leq i \leq n$. We denote by $\mathcal{C}_A$ the set of all recognized paths in $A$. From these functions, we can define the series $S_A$ associated with the automaton $A$ such that: $S_A = \sum_{w \in A^*} (S_A, w)w$, where $(S_A, w) = \bigoplus_{c \in \mathcal{C}_A, m(c) = w} \text{coef}(c)$.

We say that the automaton $A$ realizes the series $S_A$.

**Definition 8** A formal series $S \in K\langle\langle A\rangle\rangle$ is called recognizable if there exists an automaton that realizes it.

The following result due to Schützenberger [11] is classical:

**Theorem 1** (Schützenberger 1961). A formal series is recognizable if and only if it is regular.

### 3 The Position Automaton of a $K$-Expression

In order to construct a particular finite automaton, we introduce three functions over $(AE \cup TE)^*$:

\[
\text{First}(E) = \sum_{(\alpha, a_i) \in \text{pref}(E)} \text{ev}(\alpha)a_i,
\]

\[
\text{Last}(E) = \sum_{(a_i, \alpha) \in \text{suff}(E)} \text{ev}(\alpha)a_i,
\]

\[
\text{Follow}(E, a_j) = \sum_{(\alpha, a_i) \in \text{fact}(E, a_j)} \text{ev}(\alpha)a_i.
\]
with
\[
\begin{align*}
\text{pref}(E) &= \{(\alpha, a_i) \in T_E \times A_{\overline{E}} \mid \exists u \in (A_{\overline{E}} \cup T_E)^* \text{ such that } \alpha a_i u \in L(\overline{E})\}, \\
\text{suff}(E) &= \{(\alpha, a_i) \in T_E \times A_{\overline{E}} \mid \exists u \in (A_{\overline{E}} \cup T_E)^* \text{ such that } u a_i \alpha \in L(\overline{E})\}, \\
\text{fact}(E, a_j) &= \{(\alpha, a_i) \in T_E \times A_{\overline{E}} \mid \exists u, v \in (A_{\overline{E}} \cup T_E)^* \text{ such that } u \alpha a_j v \in L(\overline{E})\}.
\end{align*}
\]

For \( E = \frac{1}{2}a^* \), we get:
\[
\begin{align*}
\overline{E} &= (\frac{1}{2}a_1^*), \\
L(\overline{E}) &= \{\frac{1}{2}0^*1a_11^*, \frac{1}{2}0^*1a_11^*1^*, \cdots\}, \\
\text{First}(E) &= \frac{1}{2}a_1, \\
\text{Last}(E) &= \frac{1}{2}a_1, \\
\text{Follow}(E, a_1) &= \frac{1}{2}a_1.
\end{align*}
\]

We show now that in the case of regular \( \mathbb{K} \)-expressions the series First, Last and Follow are polynomials.

**Proposition 3** Let \( E \) be a regular \( \mathbb{K} \)-expression. Then First(\( E \)) can be inductively computed by:

\[
\begin{align*}
\text{First}(k) &= \overline{0} \text{ for all } k \in \mathbb{K}, \\
\text{First}(a) &= \alpha a_i \text{ (} a_i \text{ is the position associated to } a \text{ in } A_{\overline{E}}), \\
\text{First}(F + G) &= \text{First}(F) + \text{First}(G), \\
\text{First}(FG) &= \text{First}(F) + \text{ev}(c(F)) \text{First}(G), \\
\text{First}(F^*) &= \text{ev}(c(F))^\circ \text{First}(F).
\end{align*}
\]

**Proof.** If \( E = k \in \mathbb{K} \), one has \( L(E) = \emptyset \). Thus \( \text{pref}(E) = \emptyset \), and \( \text{First}(E) = \overline{0} \).

If \( E = a \in A \), one has \( L(a) = \{1a_1\} \). Thus \( \text{pref}(E) = \{1a_1\} \), and \( \text{First}(E) = \alpha a_1 \).

If \( E = F + G \), one has \( L(F + G) = L(F) \cup L(G) \). Thus \( \text{pref}(E) = \text{pref}(F) \cup \text{pref}(G) \), and \( \text{First}(F + G) = \text{First}(F) + \text{First}(G) \).

If \( E = FG \), one has \( L(FG) = L(F)L(G) \cup \{c(F)\}L(G) \cup L(F)\{c(G)\} \).

Thus \( \text{pref}(E) = \text{pref}(F) \cup \{c(F)\} \text{pref}(G) \), and \( \text{First}(FG) = \text{First}(F) + \text{ev}(c(F)) \text{First}(G) \).

If \( E = F^* \), one has \( L(F^*) = \{c(F)^*\} \left( \bigcup_{i \geq 1} \{L(F\{c(F)^i\})\} \right) \). Thus \( \text{pref}(E) = c(F)^* \text{pref}(F) \), and \( \text{First}(F^*) = \text{ev}(c(F))^\circ \text{First}(F) \). \( \square \)

In the case of Last and Follow, we have similar proofs.

**Proposition 4** Let \( E \) be a regular \( \mathbb{K} \)-expression. Then Last(\( E \)) can be inductively computed by:

\[
\begin{align*}
\text{Last}(k) &= \overline{0} \text{ for all } k \in \mathbb{K}, \\
\text{Last}(a) &= \alpha a_i \text{ (} a_i \text{ is the position associated to } a \text{ in } A_{\overline{E}}), \\
\text{Last}(F + G) &= \text{Last}(F) + \text{Last}(G), \\
\text{Last}(FG) &= \text{Last}(G) + \text{ev}(c(G)) \text{Last}(F), \\
\text{Last}(F^*) &= \text{ev}(c(F))^\circ \text{Last}(F).
\end{align*}
\]
Proposition 5 Let $E$ be a regular $\mathbb{K}$-expression and $x \in A_E$. Then $\text{Follow}(E, x)$ can be inductively computed by:

- $\text{Follow}(k, x) = \emptyset$ for all $k \in \mathbb{K}$,
- $\text{Follow}(a, x) = \overline{a}$ for all $a \in A$,
- $\text{Follow}(F + G, x) = \text{Follow}(F, x) + \text{Follow}(G, x)$,
- $\text{Follow}(FG, x) = \text{Follow}(F, x) + (\text{Last}(F), x) \text{First}(G) + \text{Follow}(G, x)$,
- $\text{Follow}(F^*, x) = \text{Follow}(F, x) + (\text{Last}(F^*), x) \text{First}(F)$.

Proposition 6 Let $E$ be a regular $\mathbb{K}$-expression. The formal series $\text{First}$, $\text{Last}$ and $\text{Follow}$ of $E$ are polynomials.

Proof. If $E = k$ and $E = a$, it is obvious. Suppose now that the regular $\mathbb{K}$-expressions $F$ and $G$ have finite supports for $\text{First}$, $\text{Last}$ and $\text{Follow}$. Then, by induction $F + G$, $FG$ and $F^*$ have finite supports for $\text{First}$, $\text{Last}$ and $\text{Follow}$ respectively. □

Let $h$ be the mapping from $A_\overline{E}$ to $A_E$ induced by the linearization of $E$ over $A_\overline{E}$. It maps every position to its value in $A_E$. For example, if $E = 2a_1 + 3b_2 + a_3$ then $h(a_1) = h(a_3) = a$ and $h(b_2) = b$.

Definition 9 The position automaton $A_E = (Q, q_0, F, \delta, \mu)$ of a regular $\mathbb{K}$-expression $E$ is defined by:

- $Q = A_\overline{E} \cup \{q_0\}$,
- $\delta(q_0, ah(q)) = q$ if $(\text{First}(E), q) = \alpha$,
- $\delta(p, ah(q)) = q$ if $(\text{Follow}(E, p), q) = \alpha$ for all $p, q$ in $A_\overline{E}$,
- $q \in F \subseteq Q$ if and only if $\mu(q) \neq \overline{a}$ with $\mu(q) = \begin{cases} \text{ev}(c(E)) & \text{if } q = q_0, \\ \text{Last}(E), q & \text{otherwise}. \end{cases}$

Theorem 2 Let $E$ be a regular $\mathbb{K}$-expression and $A_E$ its position automaton. Then $A_E$ realizes the regular series $S_E$.

Proof. It is shown easily that $S_{A_E} = S_E$ when $E = k$ or $E = a$. From the definition of the position automaton $A_E$, from Propositions 3, 4 and 5, and by induction on the size of the regular $\mathbb{K}$-expression $E$, one has $S_{A_{F+G}} = S_{F+G}$, $S_{A_{FG}} = S_{FG}$ and $S_{A_{F^*}} = S_{F^*}$. □

Example 1 Let $E = \left(\frac{1}{2}a^* + \frac{1}{6}b^*\right)^*$, we have:

$$\begin{align*}
\text{ev}(c(E)) &= \overline{2}, \\
\text{First}(E) &= \frac{\overline{2}}{3}a_1 + \frac{\overline{1}}{3}b_2, \\
\text{Last}(E) &= \overline{2}a_1 + \overline{2}b_2, \\
\text{Follow}(E, a_1) &= \frac{\overline{5}}{3}a_1 + \frac{\overline{1}}{3}b_2, \\
\text{Follow}(E, b_2) &= \frac{\overline{2}}{3}a_1 + \frac{\overline{4}}{3}b_2.
\end{align*}$$
The ZPC Structure with Multiplicities

The ZPC structure [12, 13] of a regular \( K \)-expression \( E \) is based on two trees deduced from the syntax tree \( T(E) \). These trees respectively encode the Last and First polynomials associated to subexpressions of \( E \). The transition relation of the position automaton \( A_E \) naturally appears as a collection of links from the Last tree to the First tree. Using this structure, we show that the computation of the transition function can be done in quadratic time in the size of \( E \). In the following, we explain how this structure is built, and we present the conversion algorithm which computes the position automaton from a regular \( K \)-expression.

A node in \( T(E) \) will be noted \( \nu \). If the arity of \( \nu \) is two, we write respectively \( \nu_g \) and \( \nu_d \) its left descendant and its right descendant. If its arity is 1, its descendant will be noted \( \nu_f \). The relation of descendence over the syntax tree is denoted \( \preceq \).

For a tree whose edges are labelled by elements of the semiring \( K \), we define the function of cost \( \pi \) as follows:

\[
\pi(\vartheta, \vartheta') = \begin{cases} 
\text{the cost of the path } \vartheta \leadsto \vartheta' \text{ if } \vartheta \preceq \vartheta' \text{ or } \vartheta' \preceq \vartheta, \\
0 & \text{otherwise}
\end{cases}
\]

We set \( \pi(\vartheta, \vartheta) = 1 \). We indicate by \( \text{Pos}(E) \) the set of positions in \( T(E) \). For a node \( \nu \) in \( T(E) \), the regular \( K \)-expression \( E_\nu \) denotes the subexpression resulting from the node \( \nu \) and \( \text{ev}(c(\nu)) \) the evaluation of its null term.

### 4.1 The Last Tree

The last tree, denoted \( TL(E) \), is deduced from a copy of the tree \( T(E) \). A node in \( TL(E) \) will be written \( \lambda \). The edges in \( TL(E) \) are directed from leaves towards the root. We will however continue to call tree the anti-tree \( TL(E) \) and we will consider the order induced by the syntax tree. The nodes of \( TL(E) \) are labeled in the following way:
Fig. 2. The Last and First trees associated with the expression $E = (\frac{1}{3}a^* + \frac{1}{6}b^*)^*$

1. For each node $\lambda$ labeled "$\cdot$", mark the edge $(\lambda, \lambda_d)$ by ev$(c(\lambda_g))$,
2. For each node $\lambda$ labeled "$\ast$", mark the edge $(\lambda_f, \lambda)$ by ev$(c(\lambda_f^*))$,
3. Any edge not marked in step 1 nor 2, will be marked by $\top$. The marking of these edges is implicit.

The node $\lambda$ represents the polynomial $e(\lambda) = \sum_{x \in \text{Pos}(E)} \pi(\lambda, x)x$.

Proposition 7 Let $E_\nu$ be the subexpression resulting from the node $\nu$. Let $\lambda$ be the node associated to $\nu$ in $\text{TL}(E)$. Then we have: $e(\lambda) = \text{Last}(E_\lambda)$ and $\pi(\lambda, x) = (\text{Last}(E_\lambda), x)$.

Proof. By induction on the size of $E$. □

4.2 The First Tree

The First tree, denoted $\text{TF}(E)$, is also deduced from the tree $T(E)$. A node in $\text{TF}(E)$ will be written $\varphi$. The edges in $\text{TF}(E)$ are directed from the root towards leaves. These edges are labelled in the following way:

1. For each node $\varphi$ labeled "$\cdot$", mark the edge $(\varphi, \varphi_d)$ by ev$(c(\varphi_g))$,
2. For each node $\varphi$ labeled "$\ast$", mark the edge $(\varphi, \varphi_f)$ by ev$(c(\varphi_f^*))$,
3. Any edge not marked in step 1 nor 2, will be marked by $\top$. The marking of these edges is implicit.

The node $\varphi$ represents the polynomial $e(\varphi) = \sum_{x \in \text{Pos}(E)} \pi(\varphi, x)x$. 
from Regular Weighted Expressions to Finite Automata

\section*{From Regular Weighted Expressions to Finite Automata}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3}
\caption{ZPC structure associated to the expression $E = \left( \frac{1}{4}a^* + \frac{1}{6}b^* \right)^*$}
\end{figure}

**Proposition 8** Let $E_\nu$ be the subexpression resulting from the node $\nu$. Let $\varphi$ be the node associated to $\nu$ in $\text{TF}(E)$. Then we have: $e(\varphi) = \text{First}(E_\varphi)$.

\subsection*{4.3 The Follow Links}

The follow links are additional links which connect the Last tree to the First tree. They are set in the following way:

1. For each node $\lambda$ labeled $".*"$, create the link $(\lambda_g, \varphi_d)$,
2. For each node $\lambda$ labeled $"*"$, create the link $(\lambda_f, \varphi)$.

If $(\lambda, \varphi)$ is a follow link, then the polynomial indicated by this link is the product of the polynomials $e(\lambda)$ and $e(\varphi)$ represented respectively by the nodes $\lambda$ and represented by the node $\varphi$. More precisely, one has $e(\lambda, \varphi) = e(\lambda)e(\varphi)$.

The set of follow links is denoted by $\Delta$.

\subsection*{4.4 Computing the Transition Function}

In this section, we show how to compute $\text{Follow}(E, x)$ which allows us to deduce the transition function $\delta$. We denote by $\Delta_x$ the set of follow links associated to the position $x$. That is: $\Delta_x = \{(\lambda, \varphi) \in \Delta | x \preceq \lambda\}$.

**Proposition 9** Let $E$ be a regular $\mathbb{K}$-expression and $x \in \text{Pos}(E)$. Then:

$$\text{Follow}(E, x) = \sum_{(\lambda, \varphi) \in \Delta_x} \pi(x, \lambda)e(\varphi)$$

(1)
As the supports of polynomials $e(\phi)$ are not disjoint, the computation of $\text{Follow}(E, x)$ according to Formula (1) requires a quadratic time. Now, we show how $\text{Follow}(E, x)$ can be computed in linear time. We introduce the function $f : \text{TF}(E) \rightarrow \text{TL}(E) \cup \{\bot\}$ defined by:

$$f(\phi) = \begin{cases} \lambda & \text{if } (\lambda, \phi) \in \Delta \times x \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ denotes an artificial node such that $\pi(x, \bot) = 0$. Let $\phi_1$ and $\phi_2$ two nodes in $\text{TF}(E)$ such that $\phi_2 \preceq \phi_1$. The polynomials induced by these two nodes are:

$$P(\phi_1) = \pi(x, f(\phi_1))e(\phi_1),$$
$$P(\phi_2) = \pi(x, f(\phi_2))e(\phi_2).$$

One has:

$$e(\phi_1) = \pi(\phi_1, \phi_2)e(\phi_2) + (e(\phi_1) - e(\phi_2)).$$

Thus $P(\phi_1)$ and $P(\phi_2)$ are deduced respectively from the polynomials:

$$P'(\phi_1) = \pi(x, f(\phi_1))(e(\phi_1) - e(\phi_2)),$$
$$P'(\phi_2) = [\pi(x, f(\phi_2)) \oplus \pi(x, f(\phi_1)) \otimes \pi(\phi_1, \phi_2)]e(\phi_2).$$

It is easy to check that $P'(\phi_1) + P'(\phi_2) = P(\phi_1) + P(\phi_2)$ and $\text{supp}(P'(\phi_1)) \cap \text{supp}(P'(\phi_2)) = \emptyset$.

Therefore, we can write

$$\text{Follow}(E, x) = \sum_{(f(\phi), \phi) \in \Delta} P(\phi) = \sum_{(f(\phi), \phi) \in \Delta} P'(\phi)$$

Algorithm 1 is based on this rewriting. It allows the computation of $\text{Follow}(E, x)$ in linear time, since a term $\alpha y$ in $\text{Follow}(E, x)$ is considered only once.

The function Delta is called with the arguments $\text{Coef} = \emptyset$ and $\phi = \phi_0$ where $\phi_0$ is the root of the tree $\text{TF}(E)$. By convention, we suppose that $\pi(\text{parent}(\phi_0), \phi_0) = 0$.

The call Delta($x$, $\bot$, $\phi_0$) is run in linear time in the size of the expression. On the one hand, it is based on a tree traversal. On the other hand, the supports of polynomials involved by the sum of Line 12 are disjoint. Then:

**Lemma 1** For all $x \in \text{Pos}(E)$, the computation of $\text{Follow}(E, x)$ requires a linear time in the size of the regular $\mathbb{K}$-expression $E$.

The final coefficients of a state $x$ is given by $\mu(x) = \pi(x, \lambda_0)$ where $\lambda_0$ is the root of the tree $\text{TL}(E)$. Thus, the final function $\mu$ can be computed in linear time using a tree traversal. From Proposition 7 and 8, and from Lemma 1, it finally comes:

**Theorem 3** Let $E$ be a regular $\mathbb{K}$-expression. The position automaton $A_E$ of $E$ can be constructed in $O(|E|^2)$ time.
Algorithm 1 Computation of $\text{Follow}(E, x)$

\begin{algorithmic}
\State $\Delta(x, \text{Coeff}, \varphi) : \text{polynomial}$
\State \textbf{Begin}
\State 1. \hspace{1em} $\text{Coeff} = \pi(x, f(\varphi)) \oplus \text{Coeff} \otimes \pi(\text{parent}(\varphi), \varphi)$
\State 2. \hspace{1em} \textbf{case} $\text{arity}(\varphi)$
\State 3. \hspace{2em} 0 :
\State 4. \hspace{3em} \textbf{if} ($\text{Coeff} \neq \overline{0}$ \text{ and } $\varphi \in \text{Pos}(E)$)
\State 5. \hspace{3em} \textbf{then}
\State 6. \hspace{4em} \textbf{return}($\text{Coeff} \varphi$)
\State 7. \hspace{3em} \textbf{else}
\State 8. \hspace{4em} \textbf{return}($\overline{0}$)
\State 9. \hspace{1em} 1 :
\State 10. \hspace{1em} \textbf{return}($\Delta(x, \text{Coeff}, \varphi_f)$)
\State 11. \hspace{1em} 2 :
\State 12. \hspace{1em} \textbf{return}($\Delta(x, \text{Coeff}, \varphi_g) + \Delta(x, \text{Coeff}, \varphi_d)$)
\State \textbf{end}
\State \textbf{End}
\end{algorithmic}

5 Conclusion

After characterizing the language associated to a $K$-expression, we have presented a new procedure to construct the position automaton of a regular weighted expression. This procedure is based on the extension of the boolean ZPC structure to the case of multiplicities. The time complexity of the construction of the position automaton is shown to be quadratic w. r. t. the size of the regular weighted expression.

References


Bideterministic Automata and Minimal Representations of Regular Languages*

Hellis Tamm and Esko Ukkonen

Department of Computer Science
P.O. Box 26, 00014 University of Helsinki, Finland
{hellis.tamm,esko.ukkonen}@cs.helsinki.fi

Abstract. Bideterministic automata are deterministic automata with the property of their reversal automata also being deterministic. It has been known that a bideterministic automaton is the minimal deterministic automaton accepting its language. This paper shows that any bideterministic automaton is the unique minimal automaton among all (including nondeterministic) automata accepting the same language. We also present a more general result that shows that under certain conditions a minimal deterministic automaton accepting some language or the reversal of the minimal deterministic automaton of the reversal language is a minimal automaton representation of the language. These conditions can be checked in polynomial time.

1 Introduction

While the minimization of a deterministic finite automaton (DFA) can be done efficiently based on the Myhill-Nerode theorem [3, Theorem 3.9], finding a minimal nondeterministic finite automaton (NFA) for a given language seems to be a more difficult problem. The decision problem of finding a minimal NFA when given a DFA of a language is a PSPACE-complete problem [4]. Furthermore, for every regular language there is a unique (up to isomorphism) minimal DFA recognizing it, whereas there may exist more than one NFA of minimal size accepting the given language. However, it is of interest to find sufficient conditions for an automaton to imply its minimality among all NFAs accepting its language.

This paper presents two minimality results. First, we show that any bideterministic automaton is the unique minimal automaton among all finite automata accepting the same language. Bideterministic automata or bideterministic languages have been considered, for example, in the context of machine learning [1], as a special case of reversible automata and languages [7], and in coding theory [8]. It has been observed that a bideterministic automaton is minimal among DFA ([1, 7]). In coding theory bideterministic trellises – which is a very restricted class of bideterministic automata – have been known to be minimal. This kind of trellises appear to correspond to certain codes (linear codes). It has been known that a minimal deterministic trellis is a minimal trellis for such codes [6]. But

* Work supported by the Academy of Finland grant 201560.
we are not aware of any such result for the general case of bideterministic automata. Second, we present some sufficient conditions under which a minimal DFA or the reversal of the minimal DFA of the reversal language is a minimal finite automaton. This is a more general result and actually, the minimality of a bideterministic automaton can be concluded from that result as well. We also show that these conditions can be tested in polynomial time.

A finite automaton is a quintuple $A = (Q, \Sigma, \delta, I, F)$ where $Q$ is a finite set of states, $\Sigma$ is the input alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. An automaton $A$ is deterministic if it has a unique initial state and if for $\forall q \in Q$ and $\forall a \in \Sigma$, $|\delta(q, a)| \leq 1$; otherwise it is nondeterministic. The reversal of an automaton $A$ is the automaton $A^R = (Q, \Sigma, \delta^R, F, I)$ where $\delta^R(p, a) = \{q \in Q \mid p \in \delta(q, a)\}$ for $\forall p \in Q$ and $\forall a \in \Sigma$. An automaton $A$ is called bideterministic if both $A$ and its reversal automaton $A^R$ are deterministic.

The empty string is denoted by $\epsilon$. For any string $x = x_1...x_k$ we denote by $x^R$ the reversal of $x$ which is the string $x_k...x_1$.

We define the extended transition function $\hat{\delta} : Q \times \Sigma^* \rightarrow 2^Q$ so that $\hat{\delta}(q, \epsilon) = \{q\}$ and $\hat{\delta}(q, xa) = \bigcup_{p \in \delta(q, x)} \delta(p, a)$ for $\forall q \in Q$, $\forall x \in \Sigma^*$ and $\forall a \in \Sigma$. A string $x \in \Sigma^*$ is accepted by $A$ if there exists $q_0 \in I$ such that $\hat{\delta}(q_0, x) \cap F \neq \emptyset$. The set $L(A) = \{x \mid \bigcup_{q \in I} \hat{\delta}(q, x) \cap F \neq \emptyset\}$ is called the language accepted by $A$. A language accepted by a bideterministic automaton is a bideterministic language. The reversal of a language $L$, denoted by $L^R$, is the set of the reversals of all the strings belonging to $L$.

A minimal automaton is an automaton with the smallest number of states among all automata that accept the given language. Two states $q_i$ and $q_j$ of $A$ are equivalent if $L((Q, \Sigma, \delta, \{q_i\}, F)) = L((Q, \Sigma, \delta, \{q_j\}, F))$. The Myhill-Nerode theorem says that a deterministic automaton is minimal among all DFAs accepting the same language if and only if no two states of it are equivalent. Two automata are equivalent if they accept the same language. Given an automaton $A$, using the well-known operation of the subset construction, we obtain an equivalent deterministic automaton $D(A) = (Q', \Sigma, \delta', q', F')$ [3, Theorem 2.1]. The automaton $D(A)$ is not necessarily the minimal DFA for $L(A)$.

2 NFA Minimization of Kameda and Weiner

Kameda and Weiner [5] have developed a theory for attacking the problem of minimization of nondeterministic automata. In the following, we present some definitions and results from this theory that we need to prove our results.

Let $A = (Q, \Sigma, \delta, I, F)$ be an automaton, $B = D(A) = (Q', \Sigma, \delta', q', F')$ and $C = D(A^R) = (Q'', \Sigma, \delta'', q'', F'')$. As $B$ and $C$ are results of the subset construction applied on the set of states $Q$ of $A$, both $Q'$ and $Q''$ consist of subsets of $Q$. 
**Definition 1.** [5, Definition 7] The states map (SM) of $A$ is a matrix which contains a row for each nonempty state of $B$, and a column for each nonempty state of $C$. The $(i,j)$ entry contains $q'_{i} \cap q''_{j}$ (or is blank if $q'_{i} \cap q''_{j} = \emptyset$) where $q'_{i}$ is the $i$-th element of $Q'$ and $q''_{j}$ is the $j$-th element of $Q''$. The elementary automaton matrix (EAM) of $A$ is obtained from the SM of $A$ by replacing each nonblank entry by a 1. Its $(i,j)$ element is denoted by $e_{ij}$.

**Theorem 1.** [5, Theorem 3] $L((Q', \Sigma, \delta', q'_{i}, F')) = \bigcup_{j | e_{ij} = 1} \{ x \in R | x \in L((Q'', \Sigma, \delta'', q''_{j}, q''_{j})) \}$.

There are observations in [5] that according to Theorem 1, any states of $B$ that have an identical pattern of 1’s and blanks in the corresponding rows of the EAM of $A$, can be merged. Also, because the definitions of $B$ and $C$ are symmetric, any states of $C$ that have the same pattern of 1’s and blanks in the corresponding columns, can be merged. These observations imply that two states of $B$ ($C$) having the same pattern of blank entries in the corresponding rows (columns) of the SM of $A$ can be merged. Rows (columns) with the same pattern of blank entries are called equivalent.

**Definition 2.** [5, Definitions 8 and 10] The reduced states map (RSM) of $A$ is obtained from the SM of $A$ by merging all equivalent rows and columns. The merging of two rows (columns) means that they are replaced by a new row (column), the entries of which are the unions of the entries of the corresponding columns (rows). The reduced automaton matrix (RAM) of $A$ is formed from the RSM of $A$ by replacing each nonblank entry with a 1.

Let $\hat{B}$ be the minimal DFA for $L(A)$, obtained from $B$ by merging by union the equivalent states, and let $\hat{C}$ be the minimal DFA, similarly obtained from $C$, for $L(C) = L(A)^{R}$.

**Lemma 1.** [5, Lemma 3] The RSM of $A$ can be obtained from $\hat{B}$ and $\hat{C}$ in the same manner as the SM of $A$ is obtained from $B$ and $C$.

**Theorem 2.** [5, Theorem 4] Equivalent automata have a RAM that is unique up to permutation of the rows and columns.

**Definition 3.** [5, Definitions 11-13] Given an EAM or RAM, if all the entries at the intersections of a set of rows $\{q'_{i1}, ..., q'_{ia}\}$ and a set of columns $\{q''_{j1}, ..., q''_{jb}\}$ are 1’s then this set of 1’s forms a grid. The grid is represented by $g = (q'_{i1}, ..., q'_{ia}; q''_{j1}, ..., q''_{jb})$. The grid $g$ contains the pair $(q'_{i}, q''_{j})$ if $i \in \{i_{1}, ..., i_{a}\}$ and $j \in \{j_{1}, ..., j_{b}\}$. A set of grids forms a cover if every 1 in the EAM (or RAM) belongs to at least one grid in the set. A minimum cover is a cover that consists of the minimum number of grids. Given a cover of an EAM (or RAM), the corresponding cover map is obtained by replacing each 1 in the EAM (or RAM) by the names of all the grids (in the given cover) it belongs to.

**Theorem 3.** [5, Theorem 5] The SM (RSM) of an automaton $A$ is a cover map, namely, the states of $A$ appear as a cover of the EAM (RAM) of $A$. 
By a special rule, an NFA can be associated with any cover of the RAM but it may be the case that this NFA is not equivalent to the original automaton. To find a minimal equivalent NFA, [5] uses an algorithm that tests the covers of the RAM in increasing order of the size to find whether the NFA for the cover is equivalent to the original automaton. The first equivalent NFA found in this way is a minimal one.

3 Bideterministic Automata Are Minimal

As we have stated in Section 1, bideterministic automata are deterministic automata with the property of their reversal automata also being deterministic. This means, among other things, that these automata have a unique initial state and a unique final state.

We also mentioned that it has been known that a bideterministic automaton is minimal among the DFAs. Indeed, to show that a bideterministic automaton is the minimal DFA for the language it accepts, one can, for example, use Brzozowski’s minimization algorithm which involves reversing, determinizing, again reversing and determinizing the automaton [2, 9]. As this algorithm, when applied to a bideterministic automaton, does not change it, it can be concluded that the automaton is minimal in the class of the DFAs. In the following we show, using the theory in Section 2, that a bideterministic automaton is also minimal in the class of the NFAs.

Let $A = (Q, \Sigma, \delta, q_0, q_f)$ be a bideterministic automaton. Its reversal automaton is $A^R = (Q, \Sigma, \delta^R, q_f, q_0)$ where $\delta^R(p, a) = \{q\} \iff \delta(q, a) = \{p\}$ for $\forall p, q \in Q$ and $\forall a \in \Sigma$. Then the automata $B$ and $C$ from Section 2 are simply $B = D(A) = A$ and $C = D(A^R) = A^R$.

Let $Q = \{q_0, ..., q_{n-1}\}$. According to Definition 1, the SM of $A$ consists of $n$ rows and $n$ columns, with exactly $n$ non-blank entries $\{q_0\}, ..., \{q_{n-1}\}$ in the matrix which are positioned so that there is exactly one such entry in every row and every column. The corresponding EAM is basically the same as SM, only these non-blank entries are replaced with 1’s. As there are no two equivalent rows nor columns in SM, it follows from Definition 2 that the RSM and RAM of $A$ are the same as SM and EAM, respectively. Now, as to Definition 3, it is not difficult to see that there are altogether $n$ different grids in EAM or RAM, with each grid consisting of just one row-column pair, and this set of grids is the only cover of the EAM or RAM. Because RAM is unique for all automata accepting $L(A)$ (Theorem 2) and any automaton accepting $L(A)$ has to have at least as many states as is the number of grids in the minimum cover of RAM (Theorem 3), we conclude that $A$ is a minimal automaton. We have proved the following theorem:

**Theorem 4.** A bideterministic automaton is minimal among all finite automata accepting the same language.

Next we show that $A$ is the unique minimal automaton accepting $L(A)$. Suppose that there exists another automaton $A'$, not isomorphic to $A$, that has $n$
states and that accepts the same language as \( A \) does. As we know that \( A \) is the only minimal deterministic automaton (because the minimal DFA is unique) then \( A' \) must be nondeterministic. This means that either \( A' \) has multiple initial states or there is a state in \( A' \) from which there are transitions with the same label to more than one state. In any case, the determinized automaton \( D(A') \) must have a state \( p - \) a subset of states of \( A' \) with the cardinality of more than one. Let \( p_1, ..., p_m \) be the states of \( A' \) comprising that subset, \( m > 1 \). Now, the row of the states map SM of \( A' \) corresponding to state \( p \) has to be such that every \( p_j, j = 1, ..., m \), belongs to at least some entry in that row (because every state of \( A' \) belongs to some state of \( D((A')^R) \)). But, as we know that the RAM of \( A' \) is the same as the RAM of \( A \) (Theorem 2), then the RAM of \( A' \) has exactly one 1 in each row and each column. This implies that the RSM of \( A' \) has exactly one non-blank entry in each row and each column. As an RSM is formed by merging the equivalent rows and columns of a SM (Definition 2), then the RSM of \( A' \) must have a row whose only non-blank entry contains all \( p_j, j = 1, ..., m \) (and possibly some other states of \( A' \)). As the RSM of \( A' \) is a cover map (Theorem 3), it has to be the case that the intersection of any two entries of the RSM is empty (otherwise the RSM of \( A' \) could not have just one non-blank entry in each row and column). Because there are \( n \) rows and \( n \) columns and thus \( n \) non-blank entries in the RSM of \( A' \), it follows that \( A' \) has at least \( n - 1 + m \) states. As we had \( m > 1 \), we get that \( A' \) must have more than \( n \) states in it. We have obtained a contradiction with the assumption we have made that \( A' \) has \( n \) states. As a conclusion we may state the following theorem:

**Theorem 5.** A bideterministic automaton is uniquely minimal.

**Remark.** In the uniqueness proof of Theorem 5 it is shown that the size of any nondeterministic automaton accepting the language of a bideterministic automaton is larger than the size of the bideterministic automaton. From this, along with the fact that a bideterministic automaton is the minimal DFA, the statement of Theorem 4 can be concluded.

**Example 1.** Let \( L_k = \{ww^R | w \in \{0,1\}^k \} \) where \( k \geq 0 \), be a set of strings consisting of concatenations of any binary string of length \( k \) and its reversal string. Let \( L^*_k \) be the set that consists of strings obtained by concatenating zero or more times the elements of \( L_k \). Then for every \( k \geq 0 \), \( L^*_k \) is accepted by a bideterministic automaton having \( 3 \times 2^k - 3 \) states; Figure 1 shows such an automaton with 9 states accepting \( L^*_2 \). By Theorem 4 we know that this is a minimal automaton recognizing this language and we cannot find a smaller automaton representation for it.

## 4 Another Minimality Result

From Theorem 4 we know that bideterminism is a sufficient condition for a language to have a property that the size of its minimal deterministic automaton is
Fig. 1. Minimal automaton of the language $L_2^*$ of Example 1

also the smallest possible size of any automaton – deterministic or nondeterministic – accepting the same language. It is of interest to find other conditions that imply similar minimality. In the following we present another, more general minimality result. Actually, Theorem 4 follows from this result as a special case.

Let $A = (Q, \Sigma, \delta, q, F)$ be a minimal deterministic automaton and let $A_1 = D(A^R) = (Q'', \Sigma, \delta'', q_1'', F'')$ be the automaton obtained from the reversal of $A$ by the subset construction. Every state $q_i''$ of $A_1$ can also be seen as a subset of the state set $Q$ of $A$.

Let $\{Q_1'', ..., Q_k''\}$ be the partition of the state set $Q''$ of $A_1$ into disjoint subsets (equivalence classes) such that any pair of states $q_i''$ and $q_j''$ of $A_1$ belongs to the same $Q_i''$, $i \in \{1, ..., k\}$, if and only if there exist states $q_{i_1}..., q_{i_l}$ of $A_1$ such that $q_{i_1}'' = q_1'', q_{i_l}'' = q_2''$ and $q_{i_j}'' \cap q_{i_{j+1}}'' \neq \emptyset$ for all $j = 1, ..., l - 1$. Moreover, if we let $Q_i = \bigcup_{q'' \in Q_i''} q''$ for $i = 1, ..., k$, then $\{Q_1, ..., Q_k\}$ forms a partition of the state set $Q$ of $A$ into disjoint subsets. Now, the following statement holds:

**Theorem 6.** Let $A$ be a minimal DFA and let $A_1 = D(A^R)$ such that either (i) every state of $A_1$ consists of at most two states of $A$, or (ii) each state of $A$ occurs in at most two states of $A_1$. Let $\{Q_1, ..., Q_k\}$ and $\{Q_1'', ..., Q_k''\}$ be the partitions of the states of $A$ and $A_1$, respectively, as described above. If $|Q_i| \leq |Q_i''|$ for all $i = 1, ..., k$, then $A$ is a minimal automaton accepting $L(A)$. If $|Q_i''| \leq |Q_i|$ for all $i = 1, ..., k$, then $A_1^R$ is a minimal automaton accepting $L(A)$.

**Proof.** Let $A = (Q, \Sigma, \delta, q, F)$ be a minimal DFA and let $A_1 = D(A^R) = (Q'', \Sigma, \delta'', q_1'', F'')$. Then the automata $B$ and $C$ of Section 2 are $B = D(A) = A$ and $C = D(A^R) = A_1$. As $B$ is equal to $A$, $B$ is the minimal DFA accepting $L(A)$. According to the Brzozowski’s minimization algorithm, $C$ is the minimal DFA
accepting $L(A^R)$ as we can write $C = D(A^R) = D(D(A)^R) = D(D((A^R)^R)^R)$.

The SM of $A$ is also the RSM of $A$ (Definitions 1 and 2, Lemma 1). The number of rows and columns in the RSM (and the RAM) of $A$ equals to the number of states of $A$ and $A_1$, respectively.

We observe that any grid in the RAM of $A$ can involve only columns that correspond to states of $A_1$ belonging to the same $Q_i^R$ for some $i \in \{1, \ldots, k\}$. This is because any two states taken from $Q_i^R$ and $Q_j^R$, where $i \neq j$, have an empty intersection and hence cannot belong to the same grid. Also, any grid in the RAM of $A$ can involve only rows that correspond to the states of $A$ belonging to the same $Q_i$ for some $i$. We say that a set of grids covers a set of rows and columns if every 1 in these rows and columns belongs to at least some grid in that grid set. For all $i = 1, \ldots, k$, let $G_i$ be any minimal set of grids that covers the set of rows and columns corresponding to $Q_i$ and $Q_i^R$, respectively. Then any $g_1 \in G_i$ and $g_2 \in G_j$, $i \neq j$, do not have a common 1 entry. It is not difficult to see that $G_1 \cup \ldots \cup G_k$ is a minimum cover of the RAM of $A$.

Now, assume that at least one of the conditions (i) and (ii) holds. This implies that every grid in the RAM of $A$ that involves several rows (columns) cannot concern more than one column (row). If this was not the case then there would be a grid $g = (q_1, \ldots, q_n; q_1', \ldots, q_n')$ in the RAM such that $a \geq 2$ and $b \geq 2$ meaning that the states $q_j$ and $q_j'$ of $A_1$ both have nonempty intersections with the states $q_1$ and $q_1$ of $A$. If (i) holds then the columns corresponding to $q_j$ and $q_j'$ must be equivalent. If (ii) holds then the rows corresponding to $q_1$ and $q_1$ must be equivalent. But neither equivalent rows nor columns can occur in RAM, so we have a contradiction. We call a grid horizontal (vertical) if it does not involve more than one row (column). So, every grid in the RAM of $A$ is either horizontal or vertical (or both if it consists of only one 1). We call a grid non-trivially horizontal (vertical) if it is horizontal (vertical) and consists of two or more 1’s.

Let us consider a set of rows and columns of the RAM of $A$, corresponding to some $Q_i$ and $Q_i^R$. We claim that a minimum number of grids that can cover these rows and columns is the minimum of $|Q_i|$ and $|Q_i^R|$. Suppose that this is not the case. Then any minimal set of grids $G_i$ covering these rows and columns must have nonempty subsets $G_i^h$ and $G_i^v$ containing non-trivial horizontal and non-trivial vertical grids, respectively. Let us assume that $G_i$ is such that any horizontal grid in $G_i$ covers the row it involves and any vertical grid in $G_i$ covers the column it involves (if this is not the case then we can find another minimal set of grids covering the same set of rows and columns which has this property).

Let us consider the case where $|Q_i| \leq |Q_i^R|$. In this case we may assume that the grids of $G_i$ are either horizontal or non-trivially vertical. The grids of $G_i^v$ must cover (in addition to $|G_i^v|$ columns) at least $|G_i^v| + 1$ rows (otherwise $G_i$ not necessarily contains non-trivial vertical grids). There remain $|Q_i^R| - |G_i^v| \geq |Q_i| - |G_i^v|$ columns which must be covered by $|G_i| - |Q_i| < |Q_i| - |G_i^v|$ horizontal grids. If (i) holds then there are exactly two 1’s in every grid of $G_i^v$ and the only way that these grids can cover $|G_i^v| + 1$ rows (they cannot cover more rows) is that they do not involve other rows than these that they cover. But this
implies \( G^R_i = G_i \), so we have obtained a contradiction. If (ii) holds then there are at most two 1’s in every row. Consider the columns that are not covered by \( G^R_i \). We had above that these columns (there are at least \(|Q_i| - |G^R_i|\) of them) must be covered by less than \(|Q_i| - |G^R_i|\) horizontal grids. This is not possible if any of these grids has only one 1. So, all of these grids must consist of two 1’s and thus they constitute \( G^R_i \). Similarly to the case (i) above, the only way that these grids can cover the columns under consideration is that they do not involve other columns than these that they cover. But this implies \( G^R_i = G_i \), so we have obtained a contradiction in this case too. The proof for the case where \(|Q'^R_i| \leq |Q_i|\) is symmetric to the proof above. The case where (i) holds is similar to the case (ii) above, and the case for (ii) is similar to the case (i) above.

So, a minimum number of grids that can cover the rows and columns of the RAM of \( A \) corresponding to some \( Q_i \) and \( Q'^R_i \), is the minimum of \(|Q_i|\) and \(|Q'^R_i|\).

We know that any automaton equivalent to \( A \) cannot have less states than is the number of grids in a minimum cover of the RAM of \( A \) (Theorems 2 and 3).

We know that \( A \) and \( A^R_1 \) both accept \( L(A) \) and their sizes are \(|Q|\) and \(|Q'^R|\), respectively. Therefore, if \(|Q_i| \leq \min|Q'_R|\) for all \( i = 1, \ldots, k \), then \( A \) is a minimal automaton accepting \( L(A) \). Also, if \(|Q'^R_i| \leq |Q_i|\) for all \( i = 1, \ldots, k \), then \( A^R_1 \) is a minimal automaton accepting \( L(A) \).

\[ \square \]

**Corollary 1.** Theorem 4 follows from Theorem 6.

**Proof.** Let \( A \) be a bideterministic automaton. Then \( A \) is a minimal DFA. The automaton \( A_1 \) of Theorem 6 is \( A^R_1 = A^R \). It is clear that both conditions (i) and (ii) of Theorem 6 hold. For all \( i = 1, \ldots, k \), \( Q_i \) and \( Q'^R_i \) consist of exactly one state, so both \(|Q_i| \leq |Q'_R|\) and \(|Q'^R_i| \leq |Q_i|\) hold. We may conclude that both \( A \) and \((A^R)^R\) are minimal automata accepting \( L(A) \). But these two automata are the same, so Theorem 4 follows. \[ \square \]

In certain cases Theorem 6 gives two different minimal automata:

**Corollary 2.** Let \( A \) and \( A_1 \) be automata as in Theorem 6. If \(|Q_i| = |Q'_R|\) for all \( i = 1, \ldots, k \), and \( A \) is not bideterministic then \( A \) and \( A^R_1 \) are two different minimal automata accepting \( L(A) \).

**Proof.** As \(|Q_i| \leq |Q'_R|\) and \(|Q'^R_i| \leq |Q_i|\) for all \( i = 1, \ldots, k \), then, according to Theorem 6, both \( A \) and \( A^R_1 \) are minimal. If \( A \) is not bideterministic then \( A^R \) has to be nondeterministic. So, \( A \) and \( A^R_1 \) must be different. \[ \square \]

**Example 2.** In Figure 2, the leftmost automaton is the minimal DFA \( A \) accepting the language \( L(00^*1 + 11^*0) \) and the rightmost one \( A_1 = D(A^R) \) is obtained from the reversal of \( A \) by the subset construction. Both conditions (i) and (ii) of Theorem 6 hold. According to the partitioning described above, the state sets of \( A \) and \( A_1 \) are partitioned into two subsets \( Q_1 = \{1, 2, 3\}, Q_2 = \{4\}, Q'_R = \)
Let us finally consider algorithms for testing the sufficient conditions for minimality given in Theorem 6. For a given minimal DFA $A$ we want to test whether or not either of the conditions (i) and (ii) of Theorem 6 is true for $D(A^R)$, and if this is the case then we want also to test whether or not either of the two alternative conditions for the sizes of the equivalence classes $Q_i$ and $Q''_i$ is true. While the subset construction that is needed to produce $D(A^R)$ is exponential in the general case, it turns out that here we can obtain a polynomial time test. This is because we can stop the construction as soon as we know that conditions (i) and (ii) cannot be true. They can be true only if $D(A^R)$ has a polynomial number of states. More precisely, the stopping criterion of the subset construction is based on the following simple lemma.

**Lemma 2.** (a) If $K$ is a collection of subsets of $\{1, 2, \ldots, n\}$ such that each member of $K$ contains at most two elements, then $|K| \leq n(n + 1)/2$. (b) If $K$ is a collection of subsets of $\{1, 2, \ldots, n\}$ such that any element of $\{1, 2, \ldots, n\}$ belongs to at most two different members of $K$, then $|K| \leq \lfloor 3n/2 \rfloor$.

**Proof.** By induction. $\Box$

**Theorem 7.** For a minimal DFA $A$ with $n$ states, one can test in time $O(n^3)$ whether or not $A$ satisfies the sufficient conditions of Theorem 6 for minimality.

**Proof.** We will describe a test algorithm and analyze its running time.

The algorithm first builds $A_1 = D(A^R)$, using the subset construction. Recall that (an incremental version of) the subset construction algorithm works by generating new candidate states for $D(A^R)$ from the already accepted states. Each new candidate is a subset of the states of $A$. The candidate is generated by putting together the states of $A$ to which $A^R$ has transitions with the same input symbol from states that constitute an already accepted state of $D(A^R)$; at the beginning the only accepted state consists of the initial states of $A^R$. 
For each new candidate the algorithm tests whether or not the candidate has already been accepted. To make this test fast we keep the already accepted states in a tree structure, called the state trie. In the state trie an accepted state \((a_1, ..., a_k)\) where \(a_1 < ... < a_k\) are some states of \(A\), in increasing order, is represented by a leaf that is reached along the path \(a_1, a_2, ..., a_k\) from the root of the trie. As the branching factor of such a trie is \(n\), we can test in time \(O(k \log n)\) whether or not a new candidate state whose \(k\) elements are given in the increasing order has already been accepted.

Now, to test the condition (i) of Theorem 6, we note first that by Lemma 2 (a) the condition can be true only if there are \(O(n^2)\) states in \(D(A^R)\). Each accepted state generates \(\leq |\Sigma|\) candidate states where \(\Sigma\) is the alphabet of \(A\) whose size is assumed to be constant. Hence the total number of candidate states is \(O(|\Sigma|n^2) = O(n^2)\). Each new candidate can be generated in time \(O(n)\) by scanning through the transition function of \(A^R\). In fact, to generate the next candidate state from a current accepted state \(X\) of \(D(A^R)\), scan the transition function and find for each \(q \in X\) the list of states that can be reached in one transition step from \(q\) with a fixed input symbol. As in our case \(|X| \leq 2\), there can be at most two such lists, each of length \(\leq n\). Then merge these lists into sorted order (to remove duplicates and to prepare for the state trie search). As \(A\) is deterministic, the size of the transition function of \(A\) and hence also of \(A^R\) is \(O(|\Sigma|n) = O(n)\). Hence a new candidate can be generated from \(X\) as described above in time \(O(n)\), and the total time for candidate generation becomes \(O(n^3)\).

If a new candidate has more than two states then we know that (i) is not true. Otherwise we next add the candidate to the state trie if it is not already there. As the height of the trie is 2, this takes time \(O(\log n)\) per candidate, hence \(O(n^2 \log n)\) altogether.

Similarly, to test the condition (ii) of Theorem 6, we note that by Lemma 2 (b) the condition can be true only if \(D(A^R)\) has \(O(n)\) states. The total number of the candidate states is therefore \(O(n)\). Each candidate can be generated in time \(O(n \log n)\). Here the factor \(\log n\) comes from merging the \(O(n)\) lists to put the elements of the candidate in the increasing order; note that an accepted state as well as a new candidate can now have \(O(n)\) elements. The total time for candidate generation becomes \(O(n^2 \log n)\). Adding a candidate to the state trie takes time \(O(n \log n)\) as the height of the trie is \(O(n)\); within the same time bound we also check that no element of the candidate occurs in more than one already accepted state. If some element already occurs in two accepted states, condition (ii) is not true. The total time for acceptance of the candidates is \(O(n^2 \log n)\).

To summarize, we have shown so far that testing (i) and (ii) can be done in time \(O(n^3)\). We have still to evaluate the sizes of classes \(Q''\) and \(Q_t\). To this end, let us construct a graph \((Q, E)\) with nodes \(Q\) and edges \(E\) where \(Q\) is the set of the states of \(A\) and \(E\) contains \((u, v)\) if and only if \(u\) and \(v\) belong to the same state of \(D(A^R)\). Set \(E\) can be built from \(D(A^R)\) in \(O(n^2)\) time. As the set of the nodes of each connected component of this graph equals some \(Q_t\), we get the sizes of the \(Q_t\)’s just by counting the number of the nodes of each.
connected component. This can be done by forming the components using the standard algorithm in time $O(n^2)$. Finally, the number of states $q''_j$ of $D(A^R)$ that together form the class $Q''_j$ and hence cover the class $Q_i$ which corresponds to our component can be evaluated combined with the construction of the component itself. We just need to associate with each node $q \in Q$ the names of all states $q''_j$ that contain $q$, and then count the total number of different names associated with a connected component. Time requirement stays $O(n^2)$. □

References

Succinct Descriptions of Regular Languages with Binary ⊕-NFAs

Lynette van Zijl

Department of Computer Science, Stellenbosch University, South Africa
lynette@cs.sun.ac.za

Abstract. Champarnaud [1] analyzed the number of states obtained from a binary ⊕-NFA during the subset construction. We extend this work to an experimental analysis of the size of the minimal DFAs obtained from binary ⊕-NFAs. We then consider the number of distinct languages accepted by binary ⊕-NFAs, and compare that to Domaratzki’s results [2] for (traditional) binary NFAs. We also show that there are certain regular languages which are accepted by succinct ⊕-NFAs, but for which no succinct traditional NFA exists.

1 Introduction

It is a well-known theoretical fact that nondeterministic finite automata (NFAs) are exponentially more succinct than deterministic finite automata (DFAs) [3], where we take the number of states as our measure of succinctness. However, this theoretical advantage of NFAs can often not be exploited in practical applications, as this theoretical bound is only achieved in quite specific situations. The question that interests us is not so much whether an NFA provides a succinct representation, but rather when (for which languages), and how often, it provides a succinct representation. Or, more formally, we are interested in the number $E_k(n, r)$ of distinct regular languages over $k$ alphabet symbols, accepted by $n$-state NFAs, for which the equivalent minimal DFA has $r$ states ($0 \leq r \leq 2^n$). Little is known about $E_k(n, r)$, even for traditional NFAs. Domaratzki [2] investigated the specific case of $E_2(n, 2^n)$ for traditional NFAs. In this paper we consider $E_2(n, 2^n)$ for symmetric difference NFAs (⊕-NFAs), and show that this value is at least as big as that for traditional NFAs.

Note that for unary ⊕-NFAs, $E_1(n, 2^n)$ was fully characterized in [4], to the extent that the precise unary regular languages for which the ⊕-NFAs offer succinct descriptions, were given. Many of these results for unary ⊕-NFAs followed by showing an equivalence between unary ⊕-NFAs and linear feedback shift registers (LFSRs) [5], used in electrical engineering and circuit design.1

* This research was supported by NRF grant #2053436.
1 The correspondence between unary ⊕-NFAs and LFSRs implies that all the known applications of LFSRs can be implemented using a variant of traditional nondeterminism. These include random number generation [6], cryptology [7], hashing, and others.

© Springer-Verlag Berlin Heidelberg 2003
The rest of this paper is organized as follows: We first define the generalization of NFAs known as $\star$-NFAs in Sect. 2. We then give an experimental analysis of the succinctness behaviour of binary $\oplus$-NFAs in Sect. 3, and in Sect. 4 we investigate $E_2(n,2^n)$ for $\oplus$-NFAs in more detail.

2 Definitions

In [6], $\star$-NFAs were defined; we briefly recap these definitions below. We assume that the reader has a basic knowledge of automata theory and formal languages, as for example in [8].

2.1 Definition of $\star$-NFAs

**Definition 1.** A $\star$-NFA $M$ is a 6-tuple $M = (Q, \Sigma, \delta, q_0, F, \star)$, where $Q$ is the finite non-empty set of states, $\Sigma$ is the finite non-empty input alphabet, $q_0 \subseteq Q$ is a set of start states and $F \subseteq Q$ is the set of final states. The transition function is given by $\delta$, such that $\delta : Q \times \Sigma \rightarrow 2^Q$, and $\star$ is any associative commutative binary operation on sets.

The transition function $\delta$ can be extended to $\delta : 2^Q \times \Sigma \rightarrow 2^Q$ by defining

$$\delta(A, a) = \bigoplus_{q \in A} \delta(q, a) \quad (1)$$

for any $a \in \Sigma$ and $A \in 2^Q$.

Also, $\delta$ can be extended to $\delta^* : 2^Q \times \Sigma^* \rightarrow 2^Q$ as follows:

$$\delta^*(A, \epsilon) = A$$

and

$$\delta^*(A, aw) = \delta^*(\delta(A, a), w)$$

for any $a \in \Sigma$, $w \in \Sigma^*$ and $A \in 2^Q$.

To obtain a $\oplus$-NFA, every occurrence of $\star$ is replaced by $\oplus$ in Definition 1 and in the extension of the transition function $\delta$:

$$\delta(A, a) = \bigoplus_{q \in A} \delta(q, a) \quad (2)$$

for any $a \in \Sigma$ and $A \in 2^Q$.

**Definition 2.** Let $M$ be a $\oplus$-NFA $M = (Q, \Sigma, \delta, q_0, F, \oplus)$, and let $w$ be a word in $\Sigma^*$. Then $M$ accepts $w$ iff $F \cap \delta(q_0, w) \neq \emptyset$.

Other possible definitions of acceptance were discussed in more detail in [9].

**Theorem 1.** Let $\mathcal{L}(M)$ be a language accepted by a $\star$-NFA $M$. Then there exists a DFA $M'$ that accepts $\mathcal{L}(M)$. 

Succinct Descriptions of Regular Languages with Binary $\oplus$-NFAs
Proof. By the well-known subset construction [8], but use (1) to calculate the transition table of the DFA. See [9] for more details.

We also make use of the following terminology in the rest of this paper:

**Definition 3.** Let $M$ be a ⊕-NFA, and let $M'$ be the DFA obtained by applying the subset construction to $M$. Then we refer to $M'$ as a ⊕-DFA.

In the rest of this paper, we loosely refer to succinct ⊕-NFAs to indicate an instance of an $n$-state ⊕-NFA for which the minimal equivalent DFA has $\Theta(f(n))$ states, where $f(n)$ is exponential in $n$.

**Example 1.** Let $M$ be a ⋆-NFA defined by

$$M = (\{q_1, q_2, q_3\}, \{a, b\}, \delta, \{q_1\}, \{q_3\}, \cdot)$$

with $\delta$ given by (see also Fig. 1)

$$\begin{array}{c|cc}
\delta & a & b \\
\hline
q_1 & \{q_2\} & \{q_1\} \\
q_2 & \{q_3\} & \{q_2\} \\
q_3 & \{q_2, q_3\} & \{q_1, q_3\} \\
\end{array}$$

If ⋆ is taken as union, $M$ is a traditional NFA, and its equivalent DFA $M' = \{Q', \{a, b\}, \delta', \{q_1\}, \{q_3\}\}$ can be obtained by the subset construction, with $\delta'$ given by

$$\begin{array}{c|cc}
\delta' & a & b \\
\hline
[q_1] & [q_2] & [q_1] \\
[q_2] & [q_3] & [q_2] \\
[q_3] & [q_2, q_3] & [q_1, q_3] \\
[q_2, q_3] & [q_2, q_3] & [q_1, q_2, q_3] \\
[q_1, q_3] & [q_2, q_3] & [q_1, q_3] \\
[q_1, q_2, q_3] & [q_2, q_3] & [q_1, q_2, q_3] \\
\end{array}$$

If $M$ were a ⊕-NFA, the subset construction must be applied using symmetric difference instead of union, and then the transition function $\delta''$ of its equivalent DFA $M''$ is given by:

Fig. 1. The ⋆-NFA for Example 1
We digress slightly to summarize the known results about the succinctness of unary $\oplus$-NFAs, as we use these results again in Sect. 4.

### 2.2 Unary $\oplus$-NFAs

It was shown in [4] that a unary $n$-state $\oplus$-NFA can be encoded as an $n \times n$ binary matrix $A$ over the Galois field $\mathbb{GF}(2)$:

For every state $q_i \in Q$, let

$$a_{ji} = \begin{cases} 1 & \text{if } q_j \in \delta(q_i, a) \\ 0 & \text{otherwise} \end{cases}$$

With each such matrix $A$ can be associated a characteristic polynomial $c(X)$ in $\mathbb{GF}(2)$, where $c(X) = \det(A - XI)$. The properties of $c(X)$ determine the cycle structure of the DFA equivalent to the unary $\oplus$-NFA. In particular, if an $n$-state unary $\oplus$-NFA $M$ is encoded as a binary matrix $A$ with characteristic polynomial $c(X)$, and $c(X)$ is primitive and irreducible over $\mathbb{GF}(2)$, then the unary DFA equivalent to $M$ has $2^n - 1$ states in a single cycle. Moreover, if the matrix $A$ is nonsingular, then all the states of the DFA lie on a number of cycles of calculable length. On the other hand, if the matrix $A$ is singular, the DFA shows transient behaviour. In this case, various sequences of states are attached to the states in the cycles in a tree-like fashion (see [4] for more detail). We illustrate these situations in the example below.

**Example 2.** Case 1: $A$ nonsingular, $c(X)$ primitive and irreducible:

Let $M$ be a unary $\oplus$-NFA defined by

$$M = (\{q_1, q_2, q_3\}, \{a\}, \delta, \{q_1\}, \{q_3\}, \oplus)$$

with $\delta$ given by

$$\begin{array}{c|cc} \delta' & a & b \\ \hline q_1 & \{q_2\} & \{q_1\} \\ q_2 & \{q_3\} & \{q_2\} \\ q_3 & \{q_1, q_3\} & \{q_1, q_3\} \\ \end{array}$$

\[2\] We assume that the reader is familiar with the theory of matrices over $\mathbb{GF}(2)$, as described for example in [10] or [11].
The equivalent DFA has a cyclic structure with \(7 = 2^3 - 1\) states in the cycle:

\[
q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow [q_1, q_3] \rightarrow [q_1, q_2, q_3] \rightarrow [q_1, q_2] \rightarrow [q_2, q_3] \rightarrow [q_1].
\]

This follows from the fact that the corresponding matrix \(A\) for \(M\) is given by

\[
A = \begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 1
\end{bmatrix}
\]

with characteristic polynomial \(c(X) = X^3 - X^2 - 1\), which is primitive and irreducible over GF(2).

Case 2: \(A\) nonsingular, \(c(X)\) not primitive and irreducible:

Let \(M\) be a \(\oplus\)-NFA defined by

\[
M = ([q_1, q_2, q_3], \{a\}, \delta, [q_1], [q_3], \oplus)
\]

with \(\delta\) given by

\[
\begin{array}{c|c}
\delta & a \\
\hline
q_1 & \{q_1, q_2\} \\
q_2 & \{q_2, q_3\} \\
q_3 & \{q_3\}
\end{array}
\]

Here, \(A\) is given by

\[
A = \begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix}
\]

The equivalent DFA is given by

\[
\begin{array}{c|c}
\delta'' & a \\
\hline
[q_1] & [q_1, q_2] \\
[q_1, q_2] & [q_1, q_3] \\
[q_1, q_3] & [q_1, q_2, q_3] \\
[q_1, q_2, q_3] & [q_1] \\
[q_2] & [q_2, q_3] \\
[q_2, q_3] & [q_2] \\
[q_3] & [q_3]
\end{array}
\]

Note that this DFA has three cycles, of lengths four, two and one (and the trivial cycle on the empty set, which we did not show). This follows from the fact that \(c(X) = (X - 1)^3\) is not primitive and irreducible. However, the matrix \(A\) is still nonsingular, and therefore the DFA does not show any transient behaviour.
Case 3: A singular:
Let $M$ be a $\oplus$-NFA defined by

$$M = (\{q_1, q_2, q_3\}, \{a\}, \delta, \{q_1\}, \{q_3\}, \oplus)$$

with $\delta$ given by

$$\begin{array}{c|c}
\delta & a \\
q_1 & \{q_1, q_2\} \\
q_2 & \{q_1, q_2\} \\
q_3 & \{q_3\} \\
\end{array}$$

The equivalent DFA is given by

$$\begin{array}{c|c}
\delta' & a \\
|q_1| & |q_1, q_2| \\
|q_1, q_2| & |q_1, q_2| \\
|q_2, q_3| & |q_1, q_2, q_3| \\
|q_1, q_2, q_3| & |q_3| \\
|q_1, q_3| & |q_1, q_2, q_3| \\
|q_2| & |q_1, q_2| \\
|q_3| & |q_3| \\
\end{array}$$

Or, in diagrammatic form,

$$\begin{array}{c|c}
\delta'' & a \\
|q_1| & |q_1, q_2| \\
|q_1, q_2| & |q_1, q_2| \\
|q_2, q_3| & |q_1, q_2, q_3| \\
|q_1, q_2, q_3| & |q_3| \\
|q_1, q_3| & |q_1, q_2, q_3| \\
|q_2| & |q_1, q_2| \\
|q_3| & |q_3| \\
\end{array}$$

Or, in diagrammatic form,

$$\begin{array}{c|c}
\delta'' & a \\
|q_1| & |q_1, q_2| \\
|q_1, q_2| & |q_1, q_2| \\
|q_2, q_3| & |q_1, q_2, q_3| \\
|q_1, q_2, q_3| & |q_3| \\
|q_1, q_3| & |q_1, q_2, q_3| \\
|q_2| & |q_1, q_2| \\
|q_3| & |q_3| \\
\end{array}$$

3 Experimental Analysis

In this section, we describe a series of experiments which examines the succinctness behaviour of binary $\oplus$-NFAs.

The experiments were conducted in the MERLin environment [12], which is based on the Grail package [13]. Each experiment consisted of the following steps:
1. For given parameters \( m \) and \( n \), a pseudo-random stream of \( m \) binary \( \oplus \)-NFAs \([6]\) were generated. Each binary \( \oplus \)-NFA contained \( n \) states, numbered from 0 to \( n - 1 \), with only one initial state (by default, state 0). The final state set contained one final state \(^3\) (by default, state \( n - 1 \)).

2. Each \( \oplus \)-NFA was converted to its equivalent DFA.

3. Each DFA was minimized.

4. Each of the resultant minimal DFAs was compared to every other DFA, and isomorphic DFAs were discarded.

5. The number of unique DFAs (and hence unique regular languages) were counted.

6. The number of states in each unique DFA was recorded.

The experiments were carried out for values of \( n \) ranging from 5 to 13. In each case, a number of small experiments \((m = 500)\), a number of medium-sized experiments \((m = 10000)\) and a number of large experiments \((m = 1000000)\) were carried out. The results obtained were similar in most cases, in the sense that the same percentage range (40% to 50%) of distinct languages were obtained. In Fig. 2, we show the results for a small experiment with parameter values \( m = 500 \) and \( n = 5 \).

From the experiments, we observed that:

- There are many binary \( \oplus \)-NFAs which have equivalent DFAs of size \( \Theta(2^n) \).
- Of the generated DFAs, many are indeed minimal.
- Of the generated DFAs, very few are isomorphic: Almost all recognize different regular languages.

The experimental results correspond to that of Champarnaud [1], in the expected number of states when applying the subset construction to a binary \( \oplus \)-NFA: Binary \( \oplus \)-NFAs tend to give rise to large DFAs. However, the experiments also indicate that many of these DFAs are indeed minimal, and, over a representative set of \( \oplus \)-NFAs, these DFAs give rise to many different regular languages.

One can explain the occurrence of a large number of DFAs with \( \Theta(2^n) \) states quite easily. For each separate alphabet symbol of the \( \oplus \)-NFA, consider the matrix representation \( A \) for the unary \( \oplus \)-NFA with only this symbol. Consider, say, alphabet symbol \( a \). If \( A \) is singular, the reachable states of the DFA, generated by \( a \), form a transient graph. However, if the matrix is nonsingular, the reachable states of the DFA for \( a \) form a number of disjoint cycles. The same holds for alphabet symbol \( b \); however, unless the transition table entries for \( a \) and \( b \) are exactly equivalent, the states reachable through \( b \) are different from those reachable through \( a \). Now, for every reachable state through \( b \), a whole cycle of new states are reached by \( a \). Provided that the cycle lengths are not trivially short (say, \( 2^{n-1} \) cycles of length 2), any reachable state from one alphabet symbol cause many more reachable states via the other alphabet symbol.

\[^3\] Experiments using a random selection of final states indicated no significant difference in the results between one final state and a randomly chosen set of final states.
In the next section, we conduct a closer investigation into the number of distinct languages accepted by binary $\oplus$-NFAs.

4 Theoretical Analysis

We are interested in the bound $G_2(n)$, which is the number of distinct languages accepted by binary $n$-state $\oplus$-NFAs. Domaratzki [2] showed that, in the case of traditional NFAs, there are at least $2^n - 2$ different languages accepted by an $n$-state binary NFA, for which the minimal equivalent DFA has $2^n$ states. We show in this section that the binary $\oplus$-NFAs perform just as well as the traditional NFAs in this regard.

The experimental results of the previous section indicate that there are very many different languages accepted by succinct binary $\oplus$-NFAs; we now set a lower bound for $E_k(n, 2^n)$ for $\oplus$-NFAs, with $k \geq 2$. The next theorem shows that, for some $n$, there are at least $2^n$ different languages accepted by succinct binary $n$-state $\oplus$-NFAs.

**Theorem 2.** For any $n$ such that $X^n - X^{n-1} - 1$ is a primitive irreducible polynomial over the Galois field GF(2), it holds that there are at least $2^n$ distinct languages accepted by an $n$-state binary $\oplus$-NFA, for which the minimal DFA accepting the language has $2^n - 1$ states.
Proof. For every valid $n$, define an $n$-state binary $\oplus$-NFA

$$M = \{q_0, \ldots, q_{n-1}\}, \{a, b\}, \delta, \{q_0\}, \{q_{n-1}\}.$$

Let $S$ be any subset of $\{q_0, q_1, \ldots, q_{n-1}\}$, and define $\delta$ as

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>${q_1}$</td>
<td>${q_0}$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${q_2}$</td>
<td>${q_1}$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$q_{n-2}$</td>
<td>${q_{n-1}}$</td>
<td>${q_{n-2}}$</td>
</tr>
<tr>
<td>$q_{n-1}$</td>
<td>${q_0, q_{n-1}}$</td>
<td>$S$</td>
</tr>
</tbody>
</table>

Since $S$ is an arbitrary subset of $Q$, this gives rise to $2^n$ different $\oplus$-NFAs.

Each of the $\oplus$-DFAs has $2^n - 1$ reachable states, since the transition function over the alphabet symbol $a$ generates a cycle of length $2^n - 1$ [10, 4].

Each of the $\oplus$-DFAs is minimal. To see this, note that the only final state in $M$ is $q_{n-1}$, so that there are exactly $f = 2^{n-1}$ final states in the DFA equivalent to $M$, and $2^n - 1 - f$ non-final states. The DFA contains a cycle of length $2^n - 1$ (on the word $a^{2^n-1}$), and hence any two given states can be equivalent only if this cycle contains a repeating pattern of final and nonfinal states. If the cycle is the repetition $p$ times of a pattern, then $p$ divides both $f$ and $f - 1$. Hence $p = 1$. Each DFA is therefore minimal.

It remains to show that each of the $\oplus$-NFAs accepts a unique language. Consider any two of the $\oplus$-NFAs, say $M_S$ and $M_{S'}$, so that $\delta_{M_S}(q_{n-1}, b) = S$ and $\delta_{M_{S'}}(q_{n-1}, b) = S'$. On the word $a^{n-1}b$, the corresponding DFAs are in the states $[S]$ and $[S']$, respectively. If $q_{n-1} \in S$ and $q_{n-1} \notin S'$, then $M_S$ accepts the word $a^{n-1}b$, but $M_{S'}$ does not.

On the other hand, suppose that both $S$ and $S'$ contains the final state $q_{n-1}$. Then there must exist some $k$, $1 \leq k \leq 2^n - 1$, such that $a^{n-1}ba^k$ is accepted by $M_S$, but not by $M_{S'}$ (otherwise, the cycle on the alphabet symbol $a$ would contain a repeating pattern).

The same argument holds if neither one of $S$ and $S'$ contains the final state.

It is known that a primitive irreducible polynomial over $\text{GF}(2)$ exists for any $n$ [5], and hence we put the following conjecture:

**Conjecture 1.** For any $n$, it holds that there are at least $2^n$ distinct languages accepted by an $n$-state binary $\oplus$-NFA, for which the minimal DFA accepting the language has $2^n - 1$ states.

To prove the conjecture, one would construct a $\oplus$-NFA as in Theorem 2 above.

The transition function on alphabet symbol $b$ will be as above, but the transition function on alphabet symbol $a$ may correspond to any irreducible primitive

---

4 We thank the anonymous referee who pointed out this proof, which is much shorter and more elegant than our original proof.
polynomial. Each of the $\oplus$-DFAs will then have $2^n - 1$ reachable states, and each would be minimal by the same arguments as in the proof above. To prove the conjecture, it remains to show that each of the $\oplus$-NFAs accepts a unique language.

We next consider the relationship between regular languages having a succinct description using traditional NFAs on the one hand, and $\oplus$-NFAs on the other hand. Let $S(n, m, \star)$ denote the set of all regular languages accepted by any $\star$-NFA with $n$ states, such that the equivalent minimal DFA has exactly $m$ states. We are interested in the relationship between $S(n, 2^n, \cup)$ and $S(n, 2^n, \oplus)$. For unary $\star$-NFAs it is known that $|S(n, 2^n, \cup)| = 0$ [14], while $|S(n, 2^n, \oplus)| = \frac{1}{n}\varphi(2^n - 1)$ [4], where $\varphi(t)$ is the Euler $\varphi$-function and denotes the number of integers less than $t$ that are relatively prime to $t$. Hence, there are regular languages which have succinct descriptions with $\oplus$-NFAs, but for which there is no succinct description with traditional NFAs. We now show that $S(n, 2^n, \cup) \cap S(n, 2^n, \oplus) \neq \emptyset$.

**Theorem 3.** Let $\Sigma$ be an alphabet of size greater than one. There is a family of languages $\{L_n\}_{n>0}$ over $\Sigma$ satisfying the following two properties:

1. $L_n$ is recognized by an $n$-state $\cup$-NFA $M$ which, when interpreted as a $\oplus$-NFA, also recognizes $L_n$, and
2. the smallest DFA recognizing $L_n$ has $\Theta(2^n)$ states.

**Proof.** Define a $\cup$-NFA $M_n = (\{0, \ldots, n-1\}, \{a, b, c\}, \delta, 0, F, \cup)$, with $F = \{0\}$ and $\delta$ given by

$$
\delta(i, a) = \begin{cases} (i + (n - 1)) \mod n & , i = 0, 1, \ldots, n - 1 \\ 1 & , i = 0 \end{cases}$$

$$
\delta(i, b) = \begin{cases} 0 & , i = 1 \\ i & , i = 2, 3, \ldots, n - 1 \\ \emptyset & , i = 0 \end{cases}$$

$$
\delta(i, c) = \begin{cases} i & , i = 1, 2, \ldots, n - 2 \\ \{0, n - 1\} & , i = n - 1. \end{cases}
$$

Now, for any subset $A$ of $\{0, \ldots, n-1\}$ it holds that $\bigcap_{j \in A} \delta(A, \sigma) = \emptyset$ for any $\sigma \in \Sigma$. Hence $M_n$ generates exactly the same DFA either as a $\cup$-NFA or as a $\oplus$-NFA.

For the $\cup$-NFA case Leiss [15] proved succinctness. Since the DFAs corresponding to the $\oplus$-NFA and the $\cup$-NFA are identical, and $F$ contains only one final state, the result also holds for the $\oplus$-NFA. □

**Corollary 1.** Suppose that $M$ is any $n$-state co-deterministic $\star$-NFA. Then there is a family of languages $\{L_n\}_{n>0}$ such that $L_n$ is recognized by the $n$-state $\cup$-NFA $M$ which, when interpreted as a $\oplus$-NFA, also recognizes $L_n$. 5

**Proof.** If $M$ is co-deterministic, then for any subset $A$ of $\{0, \ldots, n-1\}$ it holds that $\bigcap_{j \in A} \delta(A, \sigma) = \emptyset$, for any $\sigma \in \Sigma$. □

5 This corollary was pointed out by an anonymous referee.
5 Conclusion

We compared traditional NFAs and $\oplus$-NFAs, and showed that succinct binary $\oplus$-NFAs accept at least as many distinct regular languages as traditional binary NFAs. We also proved the existence of a family of languages which can be accepted succinctly by both traditional NFAs and $\oplus$-NFAs.

References

An Efficient Pre-determinization Algorithm

Cyril Allauzen and Mehryar Mohri

AT&T Labs – Research
180 Park Avenue, Florham Park, NJ 07932-0971, USA
{allauzen,mohri}@research.att.com
http://www.research.att.com/~{allauzen,mohri}

Abstract. We present a general algorithm, pre-determinization, that makes an arbitrary weighted transducer over the tropical semiring or an arbitrary unambiguous weighted transducer over a cancellative commutative semiring determinizable by inserting in it transitions labeled with special symbols. After determinization, the special symbols can be removed or replaced with ε-transitions. The resulting transducer can be significantly more efficient to use. We report empirical results showing that our algorithm leads to a substantial speed-up in large-vocabulary speech recognition. Our pre-determinization algorithm makes use of an efficient algorithm for testing a general twins property, a sufficient condition for the determinizability of all weighted transducers over the tropical semiring and unambiguous weighted transducers over cancellative commutative semirings. It inserts new transitions just when needed to guarantee that the resulting transducer has the twins property and thus is determinizable. It also uses a single-source shortest-paths algorithm over the min-max semiring for carefully selecting the positions for insertion of new transitions to benefit from the subsequent application of determinization. These positions are proved to be optimal in a sense that we describe.

1 Introduction

Weighted transducers are used in many applications such as text, speech, or image processing for the representation of various information sources [9, 11]. They are combined to create large and complex systems such as an information extraction or a speech recognition system using a general composition algorithm for weighted transducers [13].

The efficiency of such systems is dramatically increased when subsequential or deterministic transducers are used, i.e. weighted transducers with a unique initial state and with no two transitions sharing the same input label at any state. A general determinization algorithm for weighted transducers was introduced by [11]. The algorithm can be viewed as a generalization of the classical subset construction used for unweighted finite automata, it outputs a deterministic transducer equivalent to the input weighted transducer. But, unlike unweighted automata, not all weighted transducers can be determinized using that algorithm.
– this is clear since some weighted transducers do not even admit an equivalent subsequential one, they are not subsequential.

We present a general algorithm, pre-determinization, that makes an arbitrary weighted transducer over the tropical semiring or an arbitrary unambiguous weighted transducer over a cancellative commutative semiring determinizable by inserting in it transitions labeled with special symbols. After determinization, the special symbols can be removed or replaced with \( \epsilon \)-transitions. The resulting transducer can be significantly more efficient to use. We report empirical results showing that our algorithm leads to a substantial speed-up in large-vocabulary speech recognition.

Our pre-determinization algorithm makes use of an efficient algorithm for testing a general twins property [2], which is a characterization of the determinizability of functional finite-state transducers and that of unambiguous weighted automata over the tropical semiring or any cancellative commutative semiring, and also a sufficient condition for the determinizability of all weighted transducers over the tropical semiring.

The algorithm for testing the twins property determines some transitions whose presence violates the twins property. Transitions with new symbols need not be inserted at those positions however. There is some degree of freedom in the choice of those positions and their choice is critical to ensure greater benefits from the application of determinization. Our algorithm inserts new transitions just when needed to guarantee that the resulting transducer has the twins property and thus is determinizable. It uses a single-source shortest-paths algorithm over the min-max semiring for carefully selecting the positions for insertion of new transitions to benefit from the subsequent application of determinization. These positions are proved to be optimal in a sense that we describe.

2 Preliminaries

A semiring \( (\mathbb{K}, \oplus, \otimes, 0, 1) \) is a ring that may lack negation [10]. It has two associative operations \( \oplus \) and \( \otimes \) with identity elements \( 0 \) and \( 1 \). \( \oplus \) is commutative, \( \otimes \) distributes over \( \oplus \) and \( 0 \) is an annihilator for \( \otimes \). A semiring is said to be commutative when its multiplicative operation \( \otimes \) is commutative. A commutative semiring is said to be cancellative when for all \( a, b, c \) in \( \mathbb{K} \) with \( c \neq 0 \), \( a \otimes c = b \otimes c \) implies \( a = b \). The tropical semiring \( (\mathbb{R}^+ \cup \{\infty\}, \min, +, \infty, 0) \) or the real semiring \( (\mathbb{R}, +, \times, 0, 1) \) are classical examples of cancellative commutative semirings.

A weighted transducer \( T = (\Sigma, \Delta, Q, I, F, E, \lambda, \rho) \) over a semiring \( \mathbb{K} \) is an 8-tuple where \( \Sigma \) is a finite input alphabet, \( \Delta \) is a finite output alphabet, \( Q \) is a finite set of states, \( I \subseteq Q \) the set of initial states, \( F \subseteq Q \) the set of final states, \( E \subseteq Q \times \Sigma \times \Delta \times \mathbb{K} \times Q \) a finite set of transitions, \( \lambda : I \rightarrow \mathbb{K} \) the initial weight function mapping \( I \) to \( \mathbb{K} \), and \( \rho : F \rightarrow \mathbb{K} \) the final weight function mapping \( F \) to \( \mathbb{K} \) [15, 10]. Weighted automata can be defined in a similar way by simply omitting the output labels.
The results presented is this paper hold similarly for weighted transducers over the tropical semiring and unambiguous weighted transducers over a cancellative commutative semiring, cases where our algorithm for testing the twins property can be used [2]. However, to simplify and shorten the presentation, in the following, all definitions, proofs, and examples will be given for weighted transducers over the tropical semiring.

Given a transition $e \in E$, we denote by $i[e]$ its input label, $o[e]$ its output label, $w[e]$ its weight, $p[e]$ its origin or previous state and $n[e]$ its destination state or next state. Given a state $q \in Q$, we denote by $E[q]$ the set of transitions leaving $q$. A path $\pi = e_1 \cdots e_k$ in $A$ is an element of $E^*$ with consecutive transitions: $n[e_{i-1}] = p[e_i]$, $i = 2, \ldots, k$. We extend $n$ and $p$ to paths by setting: $n[\pi] = n[e_k]$ and $p[\pi] = p[e_1]$. A cycle $\pi$ is a path whose origin and destination states coincide: $n[\pi] = p[\pi]$. We denote by $P(q,q')$ the set of paths from $q$ to $q'$ and by $P(q,x,q')$ and $P(q,x,y,q')$ the set of paths from $q$ to $q'$ with input label $x \in \Sigma^*$ and output label $y$ (transducer case). These definitions can be extended to subsets $R, R' \subseteq Q$, by: $P(R,x,R') = \cup_{q \in R} P(q,x,q')$. The labeling functions $i$ (and similarly $o$) and the weight function $w$ can also be extended to paths by defining the label of a path as the concatenation of the labels of its constituent transitions, and the weight of a path as the sum of the weights of its constituent transitions: $i[\pi] = i[e_1] \cdots i[e_k]$, $w[\pi] = w[e_1] + \cdots + w[e_k]$. The weight associated by a transducer $T$ to an input string $x \in \Sigma^*$ and output string $y \in \Delta^*$ is:

$$[T](x,y) = \min_{\pi \in P(I,x,y,F)} (\lambda[p[\pi]] + w[\pi] + \rho[n[\pi]])$$  \hspace{1cm} (1)

A successful path in a weighted transducer $T$ is a path from an initial state to a final state. A state $q$ of $T$ is accessible if it can be reached from $I$. It is coaccessible if a final state can be reached from $q$. A weighted transducer $T$ is trim if it contains no transition with weight $\infty$ and if all its states are both accessible and coaccessible. $T$ is unambiguous if for any string $x \in \Sigma^*$ it admits at most one successful path with input label $x$. The inverse $T^{-1}$ of a weighted transducer $T$ is obtained by swapping the input and output labels of every transition in $T$ and its negation $-T$ by negating the cost of every transition in $T$.

The result of the composition of two weighted transducers $T_1$ and $T_2$ over the tropical semiring is the weighted transducer defined as follows. States in the composition $T_1 \circ T_2$ of $T_1$ and $T_2$ are identified with pairs of a state of $T_1$ and a state of $T_2$.\footnote{We use a matrix notation for the definition of composition as opposed to a functional notation.} Leaving aside transitions with $\epsilon$ inputs or outputs, the following rule specifies how to compute a transition of $T_1 \circ T_2$ from appropriate transitions of $T_1$ and $T_2$:\footnote{See [13] for a detailed presentation of the algorithm including the use of a filter for dealing with $\epsilon$-paths.}

$$(q_1, a, b, w_1, q'_1) \text{ and } (q_2, b, c, w_2, q'_2) \implies ((q_1, q_2), a, c, w_1 + w_2, (q'_1, q'_2))$$
When $T_2 = -T_1^{-1}$, we say that a state $(q_1, q_2)$ of the composed transducer is a diagonal state if $q_1 = q_2$. Similarly, a transition is said to be a diagonal transition when it is obtained by merging a transition $(q_1, a, b, w_1, q'_1)$ with its negative inverse $(q_1, b, a, -w_1, q'_1)$ and more generally a path is said to be a diagonal path if all its constituent transitions are diagonal.

The following definitions will also be needed in the next sections [14]. An alphabet $\Sigma$ can be extended by associating to each symbol $a \in \Sigma$ an element denoted by $a^{-1}$ and define $\Sigma^{-1}$ as: $\Sigma^{-1} = \{a^{-1} : a \in \Sigma\}$. $X = (\Sigma \cup \Sigma^{-1})^*$ is then the set of strings written over the alphabet $(\Sigma \cup \Sigma^{-1})$. If we impose that $aa^{-1}a = \epsilon$, then $X$ forms a group called the free group generated by $\Sigma$ and is denoted by $\Sigma^{(\ast)}$. Note that the inverse of a string $x = a_1 \cdots a_n$ is then simply $x^{-1} = a_n^{-1} \cdots a_1^{-1}$.

3 Determinization and the Twins Property

3.1 Determinization

A weighted automaton or transducer is said to be deterministic if it has a unique state and if no two transitions leaving the same state have the same input label. There exists a general determinization algorithm for weighted automata and transducers [11]. The algorithm is a generalization of the classical subset construction [1].

Figure 1 illustrates the determinization of a weighted automaton. The states of the output weighted automaton correspond to weighted subsets of the type $\{(q_0, w_0), \ldots, (q_n, w_n)\}$ where each $q_k \in Q$ is a state of the input machine, and $w_k$ a remainder weight. The algorithm starts with the subset reduced to $\{(p, 0)\}$ where $p$ is an initial state and proceeds by creating a transition labeled with $a \in \Sigma$ and weight $w$ leaving $\{(q_0, w_0), \ldots, (q_n, w_n)\}$ if there exists at least one state $q_k$ admitting an outgoing transition labeled with $a$, $w$ being defined by:

$$w = \min \{w_k + w[e] : e \in E[q_k], \ i[e] = a\}.$$

Similarly, Figure 2 illustrates the determinization of a finite-state transducer. Here, the states of the resulting transducer are string subsets of the type $\{(q_0, x_0), \ldots, (q_n, x_n)\}$, where each $q_k \in Q$ is a state of the input machine, and $x_k$ a remainder string. We refer the reader to [11] for a more detailed presentation of these algorithms.
Fig. 2. Determinization of finite-state transducers. (a) Finite-State transducer $T$. (b) Equivalent transducer $T'$ obtained by determinization of $T$. (c) Non-determinizable finite-state transducer, states 1 and 2 are non-twin siblings.

Unlike the unweighted automata case, not all weighted automata or finite-state transducers are determinizable, that is, the determinization algorithm does not halt with some inputs. Figure 1(c) shows an example of a non-determinizable weighted automaton and Figure 2(c) a non-determinizable finite-state transducer. Note that the automaton of Figure 1(c) differs from that of Figure 1(a) only by the weight of the self-loop at state 2. The difference between that weight and that of the similar loop at state 1 is the cause of the non-determinizability.

3.2 The Twins Property

There exists a characterization of the determinizability of weighted transducers based on a general twins property and an efficient algorithm for testing that property under some general conditions [11, 2].

The twins property was originally introduced by [6, 7, 5] to give a characterization of the determinizability of unweighted functional finite-state transducers. The definition of the twins property and the characterization results were later extended by [11] to the case of cycle-unambiguous weighted automata. The general twins property for weighted transducers presented here combines both sets of definitions and characterizations [2].

Two states $q$ and $q'$ are said to be siblings when they can be reached from the initial states $I$ by paths sharing the same input label and when there exists a cycle at $q$ and a cycle at $q'$ labeled with the same input. Figure 3(a) illustrates this definition. Two sibling states $q$ and $q'$ of a weighted finite-state transducer are said to be twins if the two following conditions:

$$o[\pi]^{-1}o[\pi'] = o[\pi c]^{-1}o[\pi' c']$$

$$w[c] = w[c']$$

hold for any paths $\pi$ from $I$ to $q$ and $\pi'$ from $I$ to $q'$, and for any cycles $c$ in $q$ and $c'$ in $q'$ such that $i[\pi] = i[\pi']$ and $i[c] = i[c']$. $T$ is said to have the twins property when it satisfies these conditions for all $q$ and $q'$. The twins property was recently shown to provide a characterization of the determinizability of all unweighted finite-state transducers [3].
Fig. 3. (a) Two sibling states \( q \) and \( q' \) in \( T \). (b) The corresponding configuration in \( -T^{-1} \circ T \)

property if any two siblings in \( T \) are twins. Note that in this definition \( q \) may be equal to \( q' \) and that we may have \( \pi = \pi' \) or \( c = c' \), or that \( \pi \) or \( \pi' \) can be the empty path if \( q \), or \( q' \), is the initial state.

In the case of weighted automata, only condition 3 on the equality of the cycle weights is required, and in the case of unweighted transducers, only condition 2 on the output labels. The twins property is a sufficient condition for the determinizability of weighted automata or weighted transducers over the tropical semiring \([11]\). It is a necessary and sufficient condition for the determinizability of unweighted transducers \([3]\) and that of unambiguous weighted automata or weighted transducers over the tropical semiring \([11, 2]\).

Polynomial-time algorithms were given by \([16, 4]\) to test the twins property for unweighted transducers. An efficient algorithm for testing the twins property for weighted and unweighted transducers was given by \([2]\). The algorithm is based on the composition of \( T \) with its negative inverse \(-T^{-1}\). Assume that \( T \) is a trim cycle-unambiguous weighted transducer over the tropical semiring, then \( T \) has the twins property if and only if the following conditions hold for any state \( q \), any path \( \pi \) from the initial state to \( q \), and any cycle \( c \) at \( q \) in \(-T^{-1} \circ T \) \([2]\):

\[
\begin{align*}
i[\pi]^{-1}o[\pi] &= i[\pi c]^{-1}o[\pi c] \quad (4) \\
w[c] &= 0 \quad (5)
\end{align*}
\]

Figures 3(a)-(b) illustrate these conditions.

4 Pre-determinization Algorithm

This section describes a general algorithm, pre-determinization, to make an arbitrary weighted transducer \( T \) over the tropical semiring or an arbitrary unambiguous weighted transducer \( T \) over a cancellative commutative semiring determinizable. The key steps of our algorithm are the following. We first augment the algorithm for testing the twins property for weighted transducers to mark the transitions of the transducer \(-T^{-1} \circ T \) that are found by the algorithm to violate the twins property with distinct markers. These markers are then used to disconnect some paths of \(-T^{-1} \circ T \) by inserting transitions with special symbols in \( T \). We use a shortest-first algorithm over a min-max algorithm to disconnect simple cycles at the best position and in the desired order of visit of the simple cycles.
### 4.1 Marking Transitions of the Composed Transducer

The algorithm for testing the twins property computes the composed transducer \( S = -T^{-1} \circ T \) and determines paths violating condition (4) or (5). We augment this algorithm to mark the transitions of \( S \) found to violate these conditions with distinct markers. More precisely, we use the following markers. If a transition \( e \) in \( S \) is marked by

i) \( M_l \), then there exist a cycle \( c \) containing \( e \) and a path \( \pi \) such that the label condition (4) does not hold;

ii) \( M_w \), then there exist a cycle \( c \) containing \( e \) and a path \( \pi \) such that the weight condition (5) does not hold;

iii) \( M_a \), then there exists a path \( \pi_0 \) containing \( e \) such that the label condition (4) does not hold for all cycles \( c \) accessible by a path \( \pi \) admitting \( \pi_0 \) as a prefix.

Markers are not exclusive, a transition may be marked with several markers or none. We denote by \( M[e] \) the set of markers assigned to a transition \( e \) by the augmented test of the twins property.

### 4.2 Disconnecting Paths

By definition of composition, a path \( \pi = e_1 \cdots e_n \) in the composed transducer \( S \) is the result of matching the input label of a path \( \pi_1 = e_1^1 \cdots e_n^1 \) of \( T \) with the input label of a path \( \pi_2 = e_1^2 \cdots e_n^2 \) of \( T \). Assume that \( \pi \) is not a diagonal path, then \( \pi \) can be eliminated from the composed machine \( S \) by inserting a new transition with a special symbol in \( \pi_1 \) or \( \pi_2 \), at any position \( i, 1 \leq i \leq n \), such that \( e_i \) is not a diagonal transition \( (e_i^1 \neq e_i^2) \), since this would prevent \( \pi_1 \) or \( \pi_2 \) to match. We then say that path \( \pi \) has been disconnected and will often use the transition \( e_i^1 \) (or \( e_i^2 \)) to refer to the position of insertion of that special transition in \( T \). The choice of the position \( e_i^1 \) (or \( e_i^2 \)) is critical for the subsequent application of determinization and will be discussed in detail in Section 4.3.

**Proposition 1 (Correctness).** Let \( T \) be a weighted transducer over the tropical semiring or an unambiguous weighted transducer over a cancellative commutative semiring, let \( S \) be the corresponding composed transducer, and let \( T' \) be the transducer obtained from \( T \) after application of the following operations:

1. if \( M[e] \cap \{ M_w, M_l \} \neq \emptyset \), disconnect all simple non-diagonal cycles containing \( e \) in \( S \).
2. if \( M[e] \cap \{ M_l \} \neq \emptyset \), disconnect all simple non-diagonal paths from an initial state leading to a diagonal cycle containing \( e \) in \( S \).
3. if \( M[e] \cap \{ M_a \} \neq \emptyset \), disconnect all simple non-diagonal cycles in \( S \) reachable from \( e \), and all simple non-diagonal paths containing \( e \) in \( S \) from the initial state to a diagonal cycle.

Then \( T' \) has the twins property and if we replace the special symbols in \( T' \) by \( \epsilon \), then \( T' \) becomes equivalent to \( T \).
Proof. The proof follows directly the definition of the twins property and the proof of the correctness of the algorithm to test for the twins property from [2].

In what follows, we will focus on the algorithm for disconnecting all the simple non-diagonal cycles containing a transition \( e \) in \( S \) with \( M[e] \cap \{ M_{w}, M_{l} \} \neq \emptyset \) (the first item of Proposition 1). A similar algorithm can be used to disconnect the paths leading to a diagonal cycle containing a transition \( e \) with \( M[e] \cap \{ M_{l} \} \neq \emptyset \) (second item). Disconnecting the paths defined by the third item of Proposition 1 can be done using the same algorithms. It first requires determining all the strongly connected components reachable from a transition \( e \) with \( M[e] \cap \{ M_{a} \} \neq \emptyset \). This can be done in time linear in the size of \( S \) by computing a topological order of the component graph of \( S \) [8].

4.3 Positions for Insertion of Transitions

As mentioned earlier, different positions can be chosen to disconnect a non-diagonal simple cycle \( C \) of \( S \). Our choice is motivated by the subsequent application of determinization, that is, we wish determinization to merge the longest possible paths to improve the efficiency of use the resulting transducer.

For any transition \( e \) in \( T \), we define its merging power, \( m[e] \), as the minimum length of the paths that can be merged with a path containing \( e \) if a special symbol is inserted at \( e \). Thus, if the choice is between two transitions \( e_{1} \) and \( e_{2} \) for the insertion of a special symbol, with \( m[e_{1}] < m[e_{2}] \), \( e_{2} \) is preferable since it can allow longer paths to be merged. We then say that \( e_{2} \) is a more favorable position for determinization than \( e_{1} \).

Since composition merges pairs of paths with matching labels, the merging power of a transition \( e \) can be naturally defined in terms of the composed transducer \( S \). Let \( E_{S} \) denote the set of transitions of \( S \) and denote by \( (e, e') \) a transition of \( E_{S} \) obtained by matching the negative inverse of the transition \( e \) and the transition \( e' \) in composition. The level of each transition \( (e, e') \in E_{S} \) in a breadth-first search tree of \( S \) can be computed in linear time in the size of \( S \) [8]. Denote by \( L[(e, e')] \) the level of \( (e, e') \). For any transition \( e \) in \( T \), let \( \Phi[e] \) be the set of non-diagonal transitions of \( E_{S} \) obtained by matching \( e \) with some other transition \( e' \). The merging power of a transition \( e \) of \( T \) can then be defined by:

\[
m[e] = \begin{cases} 
\min \{ L[(e', e'')] : (e', e'') \in \Phi[e] \} & \text{if } (\Phi[e] \neq \emptyset) \\
0 & \text{otherwise}
\end{cases}
\]

And a simple cycle \( C \) in \( S \) should be disconnected at a transition \( (e, e') \) such that \( e \) (or \( e' \)) is the most favorable position for determinization:

\[
e = \arg \max \{ m[e] : \Phi[e] \in C \}
\]

Since disconnecting one cycle may affect another, it is also important to determine in what order simple cycles are disconnected. To avoid disconnecting
Fig. 4. (a) Non-determinizable weighted automaton $A$ over the tropical semiring. The merging power $m[e]$ of each transition $e$ is indicated in square brackets. (b) Weighted automaton $B$, output of the pre-determinization algorithm applied to $A$

...
Fig. 5. The negative composition $-A^{-1} \circ A$ where $A$ is the weighted automaton of Figure 4. For each transition $(e_1, e_2)$, $\max\{m[e_1], m[e_2]\}$ is indicated in square brackets.

Fig. 6. The result of the determinization of the weighted automaton $B$ of Figure 4(b)

of $e_x$ is not along $\pi$ then $\pi$ is not disconnected and $T'$ does not have the twins property.

Example. Let $A$ be the weighted automaton over the tropical semiring shown in Figure 4(a). Figure 5 shows the composed automaton $-A^{-1} \circ A$. $A$ does not have the twins property since $-A^{-1} \circ A$ admits non-zero cycles: the cycle at state $(1, 3)$ has weight 2 and the symmetric cycle at state $(3, 1)$ has weight $-2$. The algorithm for testing the twins property marks with $M_l$ one of the transitions of each one of this cycles, e.g., the transitions from $(2, 5)$ and $(5, 2)$ labeled with $y$. A single-source shortest-distance algorithm over the min-max semiring from $(1, 3)$ determines the transition leaving state $(4, 1)$ as the position for the insertion of a special symbol since it has the largest value (3). This corresponds to inserting a new transition at the transition leaving state 4 in $A$. This insertion disconnects in fact both cycles with non-zero weight, thus no other disconnection is needed. Figure 4(b) shows $B$, the result of the application of the pre-determinization algorithm to $A$. $B$ has the twins property and is thus determinizable. Figure 6 shows the automaton obtained by determinizing $B$. 
4.4 Complexity

Let $Q$ be the set of states and $E$ the set of transitions of the weighted transducer $T$. In the worst case, the composed transducer $S = T^{-1} \circ T$ may have as many as $|Q|^2$ states and $|E|^2$ transitions. The worst-case complexity of the algorithm for testing the twins property and marking the transitions is quadratic in the size of $S$: $O(|Q|^2(|Q|^2 + |E|^2))$ [2]. The algorithm for disconnecting paths and cycles is based on a single-source shortest-distance algorithm over $(\mathbb{N} \cup \{\infty\}, \min, \max, \infty, 0)$ whose complexity is in $O(|Q|^2 \log |Q| + |E|^2)$ [12]. The algorithm also requires computing the component graph of $S$ and its topological order which can be done in linear time in the size of $S$. Thus, the overall complexity of our pre-determinization algorithm is dominated by the test of the twins property and is $O(|Q|^2(|Q|^2 + |E|^2))$.

5 Experimental Results

We have fully implemented the algorithm described in the previous sections and measured its benefits by testing it in the 5,500-word vocabulary HMIHY 0300 speech recognition task. The class-based statistical language models used in that task are not determinizable and lead to other non-determinizable machines when combined with the weighted transducers representing information sources such as the pronunciation dictionary.

Our experiments showed that our algorithm leads to a substantial recognition speed-up in this task. Figure 7 gives recognition accuracy as a function of recognition time, in multiples of real-time on a single processor of a 1GHz Intel Pentium III Linux cluster with 256 KB of cache and 2 GB of memory. Using our algorithm, the accuracy achieved by the old non-optimized integrated transducer at .4 times real-time is reached by the new system using our optimization at about .15 times real-time, that is more than 2.6 times faster.
6 Conclusion

A general algorithm was presented that makes an arbitrary weighted transducer over the tropical semiring or any unambiguous weighted transducer over a cancellative commutative semiring determinizable by inserting in it auxiliary symbols and transitions just when needed to ensure that it has the twins property. The auxiliary symbols are inserted at carefully selected positions to increase the benefits of the subsequent determinization. After determinization, the auxiliary symbols can be removed or simply replaced by the empty string.

Experiments in large-vocabulary speech recognition show that the resulting transducer can lead to a substantial recognition speed-up when the original weighted transducer is not determinizable.

References


Introducing VAUCANSON

Sylvain Lombardy¹, Raphaël Poss², Yann Régis-Gianas², and Jacques Sakarovitch³

¹ LIAFA, Université Paris 7
lombardy@liafa.jussieu.fr

² LRDE, EPITA
{yann.regis-gianas,raphael.poss}@lrde.epita.fr

³ LTCI, CNRS / ENST
sakarovitch@enst.fr

Abstract. This paper reports on a new software platform called VAUCANSON and dedicated to the computation with automata and transducers. Its main feature is the capacity of dealing with automata whose labels may belong to various algebraic structures.

The paper successively shows how VAUCANSON allows to program algorithms on automata in a way which is very close to the mathematical expression of the algorithm, describes some features of the VAUCANSON platform, including the fact that the very rich data structure used to implement automata does not weigh too much on the performance and finally explains the main choices of the programming design that enable to achieve both genericity and efficiency.

This paper reports on a new software platform dedicated to the computation with automata and transducers, which we have named VAUCANSON. The present status of VAUCANSON is still fairly experimental and many of its functions are under active development. Moreover, the user interface is still far from being friendly. However, the purposes, the design policy, and the implementation choices of the platform are well-established by now, so that it has appeared that VAUCANSON could be introduced to the community, with the undisguised aim that this software will be tried and tested and that the authors will get some feedback.

The striking feature of automata is the versatility of the concept — a labeled oriented graph — and its ability to modelize so many different kinds of machines simply by varying the domain where the labels are taken. In the most general setting, these labels are polynomials (or even rational series indeed) over a monoid M with multiplicity in a semiring K. “Classical” automata are obtained when M is a free monoid A*, when the multiplicity semiring is the Boolean semiring B and when every label is a letter in A; transducers can be

¹ Two of the authors of the paper (S. L. and J. S.) have written a \LaTeX{} macro package ([11]) that had also been coined VAUCANSON. This name has been changed into VAUCANSON–G in order to avoid confusion.

² The VAUCANSON library can be downloaded from the URL:
http://www.lrde.epita.fr/cgi-bin/twiki/view/Projects/VaucansonLib.
seen as automata over a monoid $A^* \times B^*$ with multiplicity in $B$ as well as automata over $A^*$ with multiplicity in $\mathcal{P}(B^*)$; automata over $A^*$ with multiplicity in $\mathbb{Q}$ may compute probability of occurrences of words, those with multiplicity in $(\mathbb{N}, \min, +)$ have been used in order to represent jobshop problems, etc.

Many systems already exist which manipulate automata and related structures (expressions, grammars, …) but almost all deal with automata the labels of which are letters or words — with the notable exception of FSM ([16]) which can compute with transducers and automata with “numerical” multiplicity.\(^3\)

The main idea in designing Vaucanson has been to take advantage of the most recent techniques in generic programming in order to deal with automata the labels of which may be freely chosen in any algebraic structure, with the capacity of writing independently (as far as they are independent) the algorithms on the automata on one hand and the operations in the structure on the other hand.

In the brief presentation that follows, we shall first show how the functions implemented in Vaucanson make it possible to program algorithms on automata in a way which is very close to the mathematical expression of the algorithm. The second part will describe some features of the Vaucanson platform, including the fact that the very rich data structure used to implement automata does not weigh too much on the performance. The third part explains the main choices of the programming design of the platform that enable to achieve both genericity and efficiency.

1 Writing Algorithms with Vaucanson

Another characteristic feature of automata theory, when seen from a mathematical point of view is that most statements are effective and that proofs are indeed algorithms — and in many cases, “good” proofs yield algorithms of “optimal” complexity. A first goal that is aimed with Vaucanson is to give the possibility of writing programs for algorithms on automata in a language that is as close as possible of the mathematical description of the algorithm. We illustrate this capacity on an example that is not too well-known and that we treat completely.

1.1 The Universal Automaton of a Language

The universal automaton $\mathcal{U}_L$ of a rational (regular) language $L$ is a canonical automaton. It has been (implicitly) introduced by Conway in [3] in order to solve some types of language equations. This automaton $\mathcal{U}_L$ can also be used to find the smallest NFA that accepts $L$ (cf. [1, 15]), or —as did two of the authors—to study some properties of $L$ (e.g. star height [12, 10] or reversibility [9]) at least when $L$ belongs to some subfamilies of the rational languages.

\(^3\) The FSA system ([18]) may also compute with such objects but as it is based on Prolog, the description of algorithms as well as the definition of automata is fairly different from the usage of the automata community.
The states of this automaton are the (maximal) factorizations of the language, i.e. the maximal pairs \((H, K)\) of languages such that \(H.K\) is a subset of \(L\). A state \((H, K)\) is initial (resp. final) iff the empty word belongs to \(H\) (resp. to \(K\)). There is a transition labeled by \(a\) from \((H, K)\) to \((H', K')\) iff \(H.a.K'\) is a subset of \(L\). These factorizations can be computed in the syntactic monoid, hence the universal automaton of a rational language is finite and effectively computable.

1.2 Construction of the Universal Automaton

We give here another construction (cf. [9, 19]) that does not require the computation of the syntactic monoid.

An automaton\(^4\) is defined as a 5-uple \(<Q, A, \delta, I, T>\), where \(Q\) is a finite set of states, \(A\) a finite alphabet of letters, \(\delta : Q \times A \rightarrow \mathcal{P}(Q)\) the transition function and \(I\) and \(T\) the sets of initial and final states.

Let \(D = <Q, A, \delta, \{i\}, T>\) be a deterministic automaton that accepts \(L\) (for instance, the minimal automaton of \(L\)); since \(D\) is deterministic, for every state \(p\) and every letter \(a\), \(\delta(p, a)\) is either the empty set or a singleton. The construction of \(U_L\) then goes as follow.

- Compute the co-determinized\(^5\) automaton \(C\) of the automaton \(D\). Let \(P\) be the set of states of \(C\). Every element of \(P\) is a subset of \(Q\).
- Compute the closure under intersection of the family \(P\). The result is a set \(R\): every element of \(R\) is a subset of \(Q\).
- The universal automaton is \(U_L = <R, A, \eta, J, U>\), where:
  - \(J = \{X \in R \mid i \in X\}\): \(X\) is initial iff it contains the initial state of \(D\);
  - \(U = \{X \in R \mid X \subseteq T\}\): \(X\) is final iff every element of \(X\) is final in \(D\);
  - \(\eta(X, a) = \{Y \in R \mid \forall p \in X, \delta(p, a) \cap Y \neq \emptyset\}\): there is a transition from \(X\) to \(Y\) labeled by \(a\) iff for every element of \(X\), there is a transition labeled by \(a\) to some element of \(Y\). This definition of \(\eta(X, a)\) is equivalent to:

\[
\eta(X, a) = \begin{cases}
\emptyset & \text{if } \exists p \in X, \delta(p, a) = \emptyset \\
\{Y \in R \mid \delta(X, a) \subseteq Y\} & \text{otherwise}
\end{cases}
\]

This algorithm is written in pseudo-language on Figure 1. It can be translated into a VAUCANSON function (Figure 2), that is a C++ function written with VAUCANSON macros. Notice that the variables \(J, U\) and \(\eta\) that represent initial states, final states and transitions in the pseudo-code, are useless in C++ because they are members of the automaton object. Opposite to the theoretical definition, these sets are built (both in the pseudo-language algorithm and in the VAUCANSON program) incrementally.

\(^4\) The reader is assumed to be familiar with the basic concepts and notations of automata theory, for which we essentially follow [8].

\(^5\) An automaton is co-deterministic if its transposed automaton is deterministic; the co-determinized automaton is obtained by a subset construction, like the determinized automaton.
Fig. 1. Construction of the universal automaton: The algorithm

1.3 Comments on the Code

A good understanding of this paragraph may require some knowledge about C++.

- **Vaucanson** provides a lot of new types. Every type is designed by a word ending by \textit{t}, like usual \textit{automaton t}, \textit{hstate t}, ...
- l. 3: the alias \texttt{AUTOMATON_TYPES EXACT} describes the frame in which the function will be defined and used. It fixes particularly some types. For instance, the automata we deal with here are Boolean automata on free monoid (without any multiplicity). This implies definitions of particular names for types. For instance, \texttt{automaton t} is an alias for usual \textit{automaton t}. This is the reason why usual \textit{automaton t} must be used in the declaration of function whereas, after line 3, one can declare \texttt{t}, \texttt{c} or \texttt{u} as \texttt{automaton t}. Likewise, the types \texttt{alphabet t}, \texttt{states t} are defined, and are used in some macros like \texttt{for each state} that we will explain later.
- l. 4: \texttt{d} is an automaton, \texttt{d.initial()} is the set of its initial states (which has one element, because \texttt{d} is deterministic). \texttt{d.initial().begin()} is a pointer on the first element of this set and thus \texttt{i} is the initial state of \texttt{d}.
- l. 6: It holds \texttt{co-determinized(D) = transposed(determinized(transposed(D)))}. The name of the function is \texttt{auto_transpose} to avoid confusion with the transposition (or mirror image) of words.
- l. 7: Every state of \texttt{C} is a subset of states of \texttt{D}. This relation must be made explicit: this is done with subset_\texttt{c_state}, which is a map from every state of \texttt{c} to a subset of states of \texttt{d}. This map is an optional parameter of determinize. Likewise, subset_\texttt{u_state} (line 13) is a map from every state of \texttt{u} to a subset of states of \texttt{d}.
- l. 11: The declaration of the variable \texttt{u} implies the creation of the automaton.
- l. 12: Some parameters of automata are dynamically defined. In general, it can be the monoid of labels and the semiring where multiplicities stand. This line means that these parameters are the same for \texttt{u} as for \texttt{d}. This
Fig. 2. Construction of the universal automaton: the VAUCANSON code

assignment is more an assignment of types than an usual assignement of values. In this particular case, all characteristics, except alphabet, are induced by the type automaton t (defined line 3).

- l. 14: for each(const is a macro with three parameters, the first one is a type, the third one is a container of this type and the second one is an iterator that handles the elements of that container. This line is equivalent to:

for ( pstate˙t::iterator s = u˙states.begin();
     s != u˙states.end(); s++)

- l. 15-18: For every element of the closure u˙states, a state is created and the link between the state and the corresponding subset is stored.

- l. 19: for each state is a macro; the first parameter is an iterator of states and the second one is an automaton. This line is equivalent to:

for ( state˙t::iterator x = u˙states().begin();
     x != u˙states().end(); x++)

- l. 21-24: For every state, the property of being initial or terminal is set.
– l. 25: From the automaton $u$, one can access to the “series” of $u$, and then, to the monoid on which this series is build, and, at last, to the alphabet.

– l. 28: The result of delta$_{set}$ is true if and only if, for every element $p$ of subset$_{state}[^*x]$, there exists a transition labeled by $^a$. In this case, the set of the aims of transitions labeled by $^a$ whose origin is in subset$_{state}[^*x]$ is stored in delta$_{set}$.

– l. 32: A transition from $x$ to $y$ is created, with label $a$; actually, $x$, $y$ and $a$ are iterators, and this is the reason why there is a star at the front of each of them.

2 Glimpses of the Library

The purpose of this communication is not to be a user manual of VAUCANSON and even not list all its functionalities. We give here only few hints on what is to be found in the library.

2.1 Determinization for Benchmarking

The determinization of automata (over $A^*$) is a basic algorithm found in every system. It is known that this algorithm may imply a combinatorial explosion. We compare VAUCANSON with two other systems: AMoRe [14] and FSM [16].

We consider the following family of automata: $A_n$ is an automaton with $n$ states: $\{0, 1, ..., n-1\}$ such that 0 is the only initial and the only final state, the alphabet is $\{a, b, c\}$ and the transition function $\delta$ (on the alphabet $\{a, b, c\}$) is defined by:

$$
\delta(0, a) = \{1\}, \quad \delta(0, b) = \delta(0, c) = \emptyset,$$

$$
\forall i \neq 0, \quad \delta(i, a) = \{i + 1 \mod n\},$$

$$
\delta(i, b) = \{i\}, \quad \delta(i, c) = \{0, i\}.
$$

Fig. 3. The automaton $A_n$
The determinization of the $A_n$ has been run on a Xeon 2.4Ghz, 256Ko cache memory, 1Go RAM, and yields the following table of results.

<table>
<thead>
<tr>
<th>$n$</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AMoRE</strong></td>
<td>0.02</td>
<td>0.03</td>
<td>0.13</td>
<td>0.55</td>
<td>2.62</td>
<td>12.0</td>
<td>57.4</td>
<td>*</td>
</tr>
<tr>
<td><strong>Vaucanson</strong></td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.08</td>
<td>0.39</td>
<td>1.89</td>
<td>9.08</td>
<td>43.0</td>
</tr>
</tbody>
</table>

As shown by the results, Vaucanson’s default implementation for weighted automata seems efficient even if it deals with general objects like formal series. Indeed, its design focuses on providing fast search operations. Actually, the data structure is a graph whose nodes contain many (forward and backward) pointers. As a consequence, the memory usage is higher than the one of implementations based on bitsets but the time usage is kept reasonable. Thanks to the genericity, user-defined data structures closer to the requirements of a particular application can be transparently substituted.

### 2.2 Minimization of $\mathbb{K}$-Automata

In many semirings of multiplicities, it can be hard and sometimes even impossible to find a smallest automaton that realizes a series. Yet, there exist some local conditions on the states of an automaton that allow to merge some of them. The result of this process is an equivalent $\mathbb{K}$-automaton called the minimal $\mathbb{K}$-covering (cf. [19]). This is not a canonical automaton of the series realized by the $\mathbb{K}$-automaton. Two $\mathbb{K}$-automata are bisimulation equivalent iff they have the same minimal $\mathbb{K}$-covering. This is a generalization of the well-known Nerode equivalence involved in the minimization of Boolean DFAs (e.g. see [8]). Vaucanson provides a generalized version of the Hopcroft algorithm that computes the minimal $\mathbb{K}$-covering of an automaton $A$ with multiplicity in $\mathbb{K}$.

### 2.3 From Automata to Expressions and back

Almost all systems computing with automata implement Kleene’s Theorem, that is compute a rational (regular) expression equivalent to a given automaton and conversely. Vaucanson library implements the so-called state elimination method. This method relies (as the other methods indeed) on an ordering of the states of the automaton and the expression obtained as the result depends on that ordering. A feature of the Vaucanson implementation is that the ordering is a parameter of the algorithm and can also be computed via heuristics.

The transformation of an expression into an automaton has given rise to a very rich literature. Vaucanson implements three methods: the Thompson construction, the standard automaton of an expression (also called position automaton or Glushkov automaton) and the automaton of derived terms of
an expression (also called \textit{partial derivatives} or \textit{Antimirov automaton}). For the latter, VAUCANSON implements the algorithm due to Champarnaud and Ziadi [2].

\section*{2.4 Transducer Computation}

VAUCANSON implements the two central theorems: the evaluation theorem and the composition theorem, with algorithms that correspond to the two main proof methods: the morphism realization and the representation realization and that are used according to the type of the transducers (normalized, letter-to-letter, real-time).

\section*{2.5 Programming the Algebraic Structures}

The definition of an automaton requires the definition of a semiring of multiplicities (or weights) and a monoid of labels. VAUCANSON allows the definition of any of these structures – and every generic algorithm can be applied on the resulting automata. A few of them are provided \textit{e.g.} free monoids over any finite alphabet or product of monoids; this gives access to transducers that can be considered as automata over a monoid $A^* \times B^*$. Some semirings are pre-defined too: the Boolean semiring, the usual numerical semirings (integers, floating numbers) and min-plus (or max-plus) semirings (for instance $(\mathbb{N}, \min, +)$ or $(\mathbb{Z}, \max, +)$).

The set of series over a monoid with multiplicity in a semiring is itself a semiring and can be used as such. For instance, $\text{Rat}(B^*)$ (the rational series over $B^*$ with Boolean coefficients) is a semiring and automata over $A^*$ with multiplicity in this semiring are another representation of transducers.

\section*{3 Design for Genericity}

The facilities exposed in the previous sections are not directly present in C++. A software layer is necessary to yield an abstraction level powerful enough for genericity. Yet, abstraction should not imply poor efficiency so the way of implementing polymorphism has to be carefully chosen.

This section points out the design issues involved in the development of the VAUCANSON library and its position confronted with the current known solutions of generic programming. First, we describe what helps the writing of algorithms in the framework. Then, we explain how we deal with the usual trade-off between genercity and efficiency. A new Design Pattern (cf [7]) for this purpose is presented and constitutes the contribution in the generic programming field.

\subsection*{3.1 A Unified Generic Framework}

We describe the framework for the writing of algorithm. It relies on how the object are typed, how the types of algorithm inputs are specified and how VAUCANSON can be adapted to foreign environments.
**Every Vaucanson Object Is an Element of a Set** As in Java where every entity has the Object type, every Vaucanson entity (automaton, series ...) is an instance of the \( \text{Element}_S, T \) class. \( \text{Element}_S, T \) can be read as an element of a set \( S \) implemented by the type \( T \). An instance of this class is always linked with an instance of \( S \) and an instance of \( T \). The \( S \) instance denotes the dynamic features of the set and the \( T \) instance represents the value of the element.

As a set, the \( S \) attribute represents the concept handled by the element. Since it is always linked to its set, an element can retrieve all the algebraic information it was built from. For example, \( u.set().series().monoid().alphabet() \) (Figure 2, line 25) returns the alphabet on which the automaton \( u \) is defined. This encapsulation enables shorter function prototypes and then, better readability.

Given a set \( S \), an element of \( S \) has a well-defined interface whatever its implementation. Therefore, an algorithm may involve elements implemented by different ways transparently, just by specifying that the implementations can be different. For instance, a generic algorithm which computes the product of two automata could be prototyped by:

```cpp
template <class T1, class T2>
Element<Automata,T1> product(Element<Automata,T1>, Element<Automata,T2>);
```

Finally, the implementation parameter allows a choice between different algorithm versions depending on the underlying data structure. For example, a serie can be implemented as a finite map or as a rational expression. The constant term is computed differently according to the chosen implementation.

**Focus on Weighted Automata Services** Thanks to our generic design, the design issues are centered on the general algebraic concepts (weighted automaton, general series ...). Although algebraic objects (alphabet, monoid, semiring and series) do not involve particular problems, the design decisions about the automaton object are essential according to an algorithmic and an ergonomic point of view.

As emphasized in [13], the successor function \( \delta \) is a crucial primitive because it is a general mechanism with a real algorithmic effect and which depends both on the implementation and on the concept. The \( \delta \) function must act as a glue between algorithms and data structures conveying only necessary information. Indeed, too rich a \( \delta \) can lead to inefficiency whereas too poor a \( \delta \) implies clumsy use. As a consequence, the Vaucanson library provides a large variety of \( \delta \) functions depending on algorithm needs.

The user can choose between states or edges as results. In order to obtain them, he/she also has the choice between container, output iterator or read-only access begin/end iterator couple. Finally, a criterion defines what kind of successors have to be retrieved. One can choose to return all output transitions, transitions whose labels match a particular letter or a user condition passed as a function-object. Extending Vaucanson with a new automaton implementation does not necessarily imply the definition of all these \( \delta \) functions. Indeed, many default implementations are automatically deduced from the others.
**Interaction with External Libraries** Initiated in the Standard Template Library (STL), the iterator concept is a common ground between C++ libraries. It is an abstract way of defining the generic traversal of data structures. VAUCANSON implements it to manage interoperability with STL. VAUCANSON algorithms and data structures are highly based on STL; this avoids the development of well-known structures such as list, bit vector or red-black tree.

Furthermore, importing new data structure from an external library to use it as implementation of some element can be done easily. The user has just to specify the foreign C or C++ type as implementation. Next, only a small set of external functions must be written to explain how the foreign implementation will fill the concept requirements. Then, linking with C/C++ external libraries is made natural and simple.

### 3.2 Polymorphism Using C++ Templates

Let us now introduce the considerations about genericity which have led to our framework. Object-Oriented languages enable reusability based on contracts defined by abstract classes. Yet, in practice, the late binding to abstract services is too expensive and leads too bad performance for intensive computing. The generative power of C++ template allows the static resolution of abstract services. This results in high-level C++ programs whose speed is comparable to dedicated low-level C programs. The STL has initiated this trend and its popularity demonstrates its relevancy.

**STL Approach** As mentioned in [17], the writing of generic algorithms is made easier by using primitive services common to all library data structures. For example, the iterator concept uses the presence of a begin()/end() method couple in every container to abstract its traversal. An algorithm which is generic w.r.t. the container concept is parameterized by a free type variable C. The code is written assuming that an instance of C will be a container.

Yet, parameterization à la STL does not provide any constraints to ensure that parameters really fill the requirement. Moreover, this general typing leads to overloading problems, like prototyping two algorithms with the same name and arity. As a consequence, it is not suitable for fine grained specialization. In concrete terms, this means that writing a generic algorithm for a particular class of automata is not allowed.

The main explanation is that STL lost the subclassing relation between objects because of a non constrained universal type quantification. The VAUCANSON design solves this problem by making a step further in generic programming that consists in implementing a generic object framework with static dispatch using C++-templates [6, 4]. These programming methods entail a stronger typing, which enables a finer specialization power and solves the overloading problem.
Beyond Classical Use of Templates One classical object hierarchy is not enough to obtain extensibility in our framework. The current section will describe a new design pattern we developed to allow a richer genericity. One more time, the main issue is to bound as precisely as possible the domain of an algorithm. Using only one object hierarchy would yield a one dimensional discrimination. Yet, a fine grained specialization would require the hierarchy to be a directed acyclic graph (with multiple inheritance).

To simplify the object organization, we define more components to characterize an object. We notice that abstraction and implementation are quite orthogonal for at least two reasons. Firstly, when writing a general algorithm, people should only focus on the mathematical concepts. Implementation constraints are taken into account afterwards. Second, algorithm specialization should depend on implementation and concept symmetrically. Because of this orthogonality, it is easier to design the implementation and the concept separately. Design patterns for this purpose are the classical BRIDGE [7] or more recently the GENERIC BRIDGE [5]. However, there remain two problems for us: first, it is asymmetric, privileging concept upon implementation; second, it does not allow subclassing w.r.t the two parameters because template arguments are invariant. To solve all these problems, the VAUCANSON library uses a new design pattern called ELEMENT/METAELEMENT. The main idea is to enable de-construction of an object w.r.t its two components and to use them for typing purpose. Element is a generic class associating a concept class and an implementation one. The role of MetaElement is to define the interaction between these two components that is, how the data structure implements the concept. A kind of multi-methods with static dispatch is also used to allow default implementation and specialization of n-ary methods. The pattern is illustrated in the figure 4 using the Unified Modelling Language. Its effective implementation involves some C++ meta-programming techniques which will not be explicit in this paper.
Introducing Vaucanson

References


WFSC – A New Weighted Finite State Compiler

André Kempe¹, Christof Baejis¹, Tamás Gaál¹, Franck Guigne¹,², and Florent Nicart¹,²

¹ Xerox Research Centre Europe, Grenoble Laboratory
  6 chemin de Maupertuis, 38240 Meylan, France
  firstname.lastname@xrce.xerox.com
  http://www.xrce.xerox.com

² Laboratoire d’Informatique Fondamentale et Appliquée de Rouen
  Faculté des Sciences et des Techniques, Université de Rouen
  76821 Mont-Saint-Aignan, France
  firstname.lastname@dir.univ-rouen.fr
  http://www.univ-rouen.fr/LIFAR/

Abstract. This article presents a new tool, WFSC, for creating, manipulating, and applying weighted finite state automata. It inherits some powerful features from Xerox’s non-weighted XFST tool and represents a continuation of Xerox’s work in the field of finite state automata over two decades. The design is generic: algorithms work on abstract components of automata and on a generic abstract semiring, and are independent of their concrete realizations. Applications can access WFSC’s functions through an API or create automata through an end-user interface, either from an enumeration of their states and transitions or from rational expressions.

1 Introduction

Finite state automata (FSAs) are mathematically well defined and offer many practical advantages. They allow for fast processing of input data and are easily modifiable and combinable by well defined operations. Therefore, FSAs are widely used in Natural Language Processing (NLP) (Kaplan and Kay, 1981; Koskenniemi, Tapanainen, and Voutilainen, 1992; Sproat, 1992; Karttunen et al., 1997; Mohri, 1997; Roche and Schabes, 1997; Sproat, 2000) and in many other fields. There are several toolkits that support the creation and use of FSAs, such as XFST (Karttunen et al., 1996-2003; Beesley and Karttunen, 2003), FSA Utilities (van Noord, 2000), FIRE Lite (Watson, 1994), INTEX (Silberztein, 1999), and many more.

Weighted finite state automata (WFSAs) combine the advantages of ordinary FSAs with those of statistical models, such as Hidden Markov Models (HMMs), and hence have a potentially wider scope of application than FSAs. Some toolkits support the work with WFSAs, such as the pioneering implementation FSM (Mohri, Pereira, and Riley, 1998), Lextools on top of FSM (Sproat, 2003), and FSA Utilities (van Noord, 2000).
WFSC (Weighted Finite State Compiler) is our new tool for creating, manipulating, and applying WFSAs. It inherits some powerful features from Xerox’s non-weighted XFST tool, that are crucial for many practical applications. For example, the “unknown symbol” allows us to assign the infinite set of all unknown symbols to a single transition rather than declaring in advance all symbols that potentially could occur and assigning each of them to a separate transition. This saves a considerable amount of memory and processing time. Flag diacritics, another feature proposed by Xerox, can also reduce the size of FSAs. They are extensively used in the analysis of morphologically rich languages such as Finnish and Hungarian. WFSC represents a continuation of Xerox’s work in the field of FSAs, spanning over two decades (Kaplan and Kay, 1981; Karttunen, Kaplan, and Zaenen, 1992; Karttunen et al., 1996-2003; Beesley and Karttunen, 2003).

This article is structured as follows: Section 2 explains some of the mathematical background of WFSAs. Section 3 gives an overview of the modular generic design of WFSC, describing the system architecture (3.1), the central role and the implementation of sets (3.2), and the approach for programming the algorithms (3.3). Section 4 presents WFSC from the users’ perspective, describing the end-user interface (4.1) and an example of application (4.2). Section 5 concludes the article.

2 Preliminaries

In this section we recall the basic definitions of our framework: algebraic structures such as monoid and semiring, as well as weighted automata and transducers (Eilenberg, 1974; Kuich and Salomaa, 1986).

2.1 Semirings

A monoid consists of a set \( M \), an associative binary operation \( \circ \) on \( M \), and a neutral element \( \bar{1} \) such that \( \bar{1} \circ a = a = a \circ \bar{1} \) for all \( a \in M \). A monoid is called commutative iff \( a \circ b = b \circ a \) for all \( a, b \in M \).

The set \( K \) with two binary operations \( \oplus \) and \( \otimes \) and two elements \( \bar{0} \) and \( \bar{1} \) is called a semiring, if it satisfies the following properties:

1. \( \langle K, \oplus, \bar{0} \rangle \) is a commutative monoid
2. \( \langle K, \otimes, \bar{1} \rangle \) is a monoid
3. \( \otimes \) is left- and right-distributive over \( \oplus \):
   \[ a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c), \quad (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c), \quad \forall a, b, c \in K \]
4. \( \bar{0} \) is an annihilator for \( \otimes \):
   \[ \bar{0} \otimes a = a \otimes \bar{0} = \bar{0}, \quad \forall a \in K \]

We denote a generic semiring \( K \) as \( \langle K, \oplus, \otimes, \bar{0}, \bar{1} \rangle \).

Some automaton algorithms require semirings to have specific properties. For example, composition as proposed by (Pereira and Riley, 1997; Mohri, Pereira, and Riley, 1998) requires a semiring to be commutative, and \( \varepsilon \)-removal as proposed by (Mohri, 2002) requires it to be \( k \)-closed. These properties are defined as follows:
1. commutativity: \( a \otimes b = b \otimes a , \forall a, b \in K \)
2. \( k \)-closedness: \( \bigoplus_{n=0}^{k+1} a^n = \bigoplus_{n=0}^k a^n , \forall a \in K \)

The following well-known examples are all commutative semirings:

1. \( \langle \mathcal{B}, +, \times, 0, 1 \rangle \): boolean semiring, with \( \mathcal{B} = \{0, 1\} \) and \( 1 + 1 = 1 \)
2. \( \langle \mathbb{N}, +, \times, 0, 1 \rangle \): integer semiring with the usual addition and multiplication
3. \( \langle \mathbb{R}^+, +, \times, 0, 1 \rangle \): real positive sum times semiring
4. \( \langle \mathbb{R}, \min, +, \infty, 0 \rangle \): a real tropical semiring where \( \mathbb{R}^+ \) denotes \( \mathbb{R}^+ \cup \{\infty\} \)

A number of algorithms require semirings to be equipped with an order or partial order denoted by \( \prec_K \) (example in Section 3.3). Each idempotent semiring \( K \) (i.e., \( \forall a \in K : a \oplus a = a \)) has a natural partial order defined by \( a \prec_K b \iff a \oplus b = a \). In the above examples, the boolean and the real tropical semiring are idempotent, and hence have a natural partial order.

### 2.2 Weighted Automata and Transducers

A weighted automaton \( A \) over a semiring \( K \) is defined by the 6-tuple \( \langle \Sigma, Q, I, F, E_A, K \rangle \), and a weighted transducer \( T \) by the 7-tuple \( \langle \Sigma, \Omega, Q, I, F, E_T, K \rangle \), where

- \( \Sigma, \Omega \) are finite alphabets
- \( Q \) is the finite set of states
- \( I \subseteq Q \) is the set of initial states
- \( F \subseteq Q \) is the set of final states
- \( E_A \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q \) is the set of transitions of \( A \)
- \( E_T \subseteq Q \times \Sigma \cup \{\varepsilon\} \times \Omega \cup \{\varepsilon\} \times Q \) is the set of transitions of \( T \)
- \( K \) is a semiring

In the following, both automata and transducers will be referred to as networks. By convention, our networks have only one initial state \( i \in I \) without loss of generality since for any network with multiple start states there exists a network with a single start state accepting the same language. For any state \( q \in Q \), we denote by

- \( \lambda(q) : I \to K \) the initial weight function with \( \lambda(q) = 0 \), \( \forall q \notin I \)
- \( \varrho(q) : F \to K \) the final weight function with \( \varrho(q) = 0 \), \( \forall q \notin F \)

and for any transition \( e \in E \)

- \( w(e) : W \to K \) the weight of \( e \) with \( w(e) \neq 0 \), \( \forall e \in E \)
- \( p(e) : P \to Q \) the source state of \( e \)
- \( n(e) : N \to Q \) the target state of \( e \)
- \( a(e) : A \to \Sigma \cup \{\varepsilon\} \times \Omega \cup \{\varepsilon\} \) the label of \( e \)
A path $\pi$ of length $l = |\pi|$ is a sequence of transitions $e_1,e_2,\ldots,e_l$ such that $n(e_i)=p(e_{i+1})$ for all $i \in [1,l-1]$. A path is said to be successful iff $p(e_1) \in I$ and $n(e_l) \in F$. For any successful path $\pi$, the accepting weight $w(\pi)$ is given by

$$w(\pi) = \lambda(p(e_1)) \otimes \left( \bigotimes_{j=1}^{l} w(e_j) \right) \otimes \rho(n(e_l)) \quad (1)$$

We denote by $\Pi(s)$ the set of successful paths for the input string $s$. Thus, the accepting weight for any input string $s$ is defined by

$$w(s) = \bigoplus_{\pi \in \Pi(s)} w(\pi) \quad (2)$$

Composition of transducers $T_i$ is expressed either by the $\circ$ or the $\odot$ operator. However, $T_1 \circ T_2 = T_2 \circ T_1$ which corresponds to $T_2(T_1(\ ))$ in functional notation (Birkhoff and Bartee, 1970).

3 Modular Generic Design

3.1 Layers of the WFSC Library

WFSC has been designed in a modular and generic way in several layers to facilitate its maintenance and the implementation of new algorithms (Figure 1):

Fig. 1. Layers of the WFSC library (simplified extract)
bottom layer contains different physical realizations of automaton components such as many different types of states and transitions. It is followed by a layer of abstract automaton components such as one single abstract type of transitions or states. The next higher layer contains basic automaton algorithms, and is followed by a layer of more complex algorithms. Algorithms work only on abstract components, and are independent of their concrete physical realizations in the lowest layer. Physical components can change their form (to optimally adapt to a situation) without disturbing the correct functioning of the algorithms. C++ was chosen as the implementation language, as a compromise between expressive power, efficiency, and modularity.

Semirings, which are themselves modular theoretical concepts, are implemented in WFSC in a modular way. An algorithm always works on a generic (abstract) semiring, and functions correctly no matter which actual semiring is behind it (provided the semiring has all properties required by the algorithm).

Programmers of algorithms will work only with abstract automaton components (low-level interface) and do not have to deal with their concrete physical realizations. Programmers of practical applications can use WFSC’s function library through an API or an end-user interface (Section 4.1).

3.2 Central Role of Sets

Sets play an important role in automata theory. An automaton has a state set $Q$ and a transition set $E$ and each of its states $q \in Q$ has a set of outgoing transitions $E(q)$. Automaton algorithms make extensive use of set operations. In addition to the above sets they may manipulate sets of auxiliary "objects" such as state pairs or pairs of a state and some related weight. For example, the pseudocode in Figure 3, showing a modified version of the Viterbi algorithm (Viterbi, 1967), contains 7 out of 18 lines with set operations (lines: 4, 6, 7, 8, 12, 15, 17)

To facilitate an efficient and easy implementation of algorithms, we consider it crucial to provide WFSC with a generic and flexible implementation of sets supporting a large number of basic set operations, alternative internal structures such as vector, list, binary tree, etc. We designed sets so that they have a compact representation since each of the (possibly many millions of) states of an automaton has a set of outgoing transitions, and since algorithms like composition can dramatically increase that number.

Since the default implementation of sets in C++ does not meet these requirements, a special technique has been developed, called Bitwise Virtuality, that allows class abstraction and polymorphism at little cost in memory (Nicart, 2003). Furthermore, this mechanism allows on-the-fly changing of type and methods of existing objects and hence on-the-fly conversion among set structures for runtime optimization in different steps of an algorithm.

3.3 Algorithm Programming Style

The layer of abstract automaton components (low-level interface, Figure 1) in the WFSC library allows us to write algorithms in a style close to pseudocode.
Low-level operations (such as keeping a list linked) are hidden inside the C++ classes that implement the abstract components. This approach facilitates the implementation and maintenance of algorithms by allowing programmers to fully concentrate on the algorithms themselves rather than on low-level operations. Numerous versions of a new algorithm can be tested in relatively short time.

We illustrate this programming style on the example of a modified version of the Viterbi algorithm. The “classical” Viterbi algorithm is used for estimating in linear time the most likely path through a Hidden Markov Model (HMM), given a sequence of observation symbols (Viterbi, 1967; Rabiner, 1990; Manning and Schütze, 1999).

We use our modified version of the algorithm for identifying the “best” path of bounded length in a WFSA, ignoring the symbols. Conceptually, the algorithm uses a trellis of nodes where each row corresponds to one state in the WFSA, each column to one step in the traversal (Figure 3). For example, the node in row 2 and column 3 represents the fact of reaching state 2 after traversing 3 transitions.

In the pseudocode of the algorithm, we describe each trellis node (in column $t$) by a 4-tuple $m_t = (q_t, \psi_t, e_{t-1}, m_{t-1})$ with $q_t$ being the state of $m_t$, $\psi_t$ the weight of the best path from the initial state $i$ to $q_t$, $e_{t-1}$ the last transition on this path, and $m_{t-1}$ a back-reference to the trellis node of the source state of $e_{t-1}$ (Figure 2). Absent elements are expressed by $\perp$. The sets $M_t$ and $M_{t+1}$ describe the columns $t$ and $t+1$ respectively. Given two weights $w$ and $w'$, we write $w \succ w'$ to express that $w$ is “better” or “worse” than $w'$ respectively, meaning $w > w'$ or $w < w'$ according to maximum or minimum search.

The algorithm first checks a property of the semiring $K$, namely whether continuing on a path $\pi$ beyond some state $q$ can lead to a better weight than the one compiled up to $q$ (line 1). It then initializes $M_0$ with a single $m_0$ corresponding to the initial state $i$ (Figure 3b and Figure 2 lines 3,4). It inserts into each following set $M_{t+1}$ one element $m_{t+1}$ for each state $q'$ that can be reached by a transition $e$ from some state $q$ having an element $m_t \in M_t$ (lines 8 to 17): each newly created $m_{t+1}$ (line 14) is compared to a previously created $m'_{t+1}$ of the same state $n(e)$ which either exists in $M_{t+1}$ or is $\perp$ (lines 15,16). Only the best of the two is kept in $M_{t+1}$ (lines 16,17) so that at any time there is at most one $m_{t+1}$ for a given state $q'$ in $M_{t+1}$. A transition $e$ is not taken if it cannot be on the best path (lines 1,13). Whether an $m$ is better than another one depends on the following conventions: $\psi(m) \succ \psi(m') \Rightarrow m \succ m'$, $m = \perp \Rightarrow (\psi(m) = 0 \land g(q(m)) = 0)$, and $(m \neq \perp \land m' = \perp) \Rightarrow m \succ m'$. When a final state $q_t$ is reached, we have identified a complete path whose weight is $\psi_t \otimes g(q_t)$ (since $\lambda(i) = 1$). The variable $\hat{m}$ refers to the $m$ of the final state $q$ of the best complete path found so far (lines 9 and 10).

In the C++ program, $m_t$ is represented by $m_0 = \{q, \psi, e_{\text{prev}}, m_{\text{prev}}\}$, $m_{t+1}$ by $m_1$, $m'_{t+1}$ by $m_1$, and $\hat{m}$ by $m_{\text{Best}}$. The sets $M_t$ and $M_{t+1}$ are denoted by $M_0$ and $M_1$ respectively and use our own implementation of sets (Section 3.2). Null pointers indicate absent elements.
ViterbiBestPath(A, maxlength) :
1 \kappa \leftarrow (\forall w_1, w_2 \in K: w_1 \otimes w_2 \neq w_1) \quad \text{(improvement impossible)}
2 \vec{m} \leftarrow \bot
3 m_0 \leftarrow \langle i, \vec{1}, \bot, \bot \rangle
4 M_0 \leftarrow \{m_0\}
5 t \leftarrow 0
6 \text{while } (t \leq \text{maxlength}) \land (M_t \neq \emptyset) \text{ do}
7 M_{t+1} \leftarrow \emptyset
8 \quad \text{for } \forall m_t \in M_t \text{ do}
9 \quad \quad \text{if } \psi(\vec{m}) \otimes g(q(\vec{m})) < \psi(m_t) \otimes \rho(q(m_t))
10 \quad \quad \quad \text{then } \vec{m} \leftarrow m_t
11 \quad \quad \text{if } t < \text{maxlength}
12 \quad \quad \quad \text{for } \forall e \in E(q_t) \text{ do}
13 \quad \quad \quad \quad \text{if } \neg (\kappa \land ((p(e) = n(e)) \lor (g(p(e)) > w(e))))
14 \quad \quad \quad \quad \text{then } m_{t+1} \leftarrow \langle n(e), \psi_t \otimes w(e), e, m_t \rangle
15 \quad \quad \quad \quad \text{if } m_{t+1} < m_{t+1}
16 \quad \quad \quad \quad \quad \text{then } M_{t+1} \leftarrow \{M_{t+1} - \{m_{t+1}\}\} \cup \{m_{t+1}\}
17 \quad \quad \text{t} \leftarrow t+1
18 \quad \text{return BuildPath(\vec{m})}

Wfsa ViterbiBestPath (Wfsa* A, int maxlength, bool min_search)
{
bool (*better_weight)(Weight, Weight, Semiring*) = (min_search) ? lower_weight : higher_weight;
Set<m> M0(), M1();
......;
1: bool improvement_imposs = (min_search) ? A->K->is_monAscending() : A->K->is_monDescending();
2: m* mBest = 0;
3: m* m0 = new m (A->i, A->K->_1, 0, 0);
4: M0.insert (m0);
5: int t = 0;
6: \text{while } ((t \leq \text{maxlength}) \&\& (M0.size() > 0)) {
7: \text{M0.clear();}
8: \text{for (M0_Iterator.connect (M0); !M0_Iterator.end(); M0_Iterator++) \{}
9: \text{n0 = M0_Iterator.item();}
10: \text{if (better_n (n0, mBest, A->K, better_weight))}
11: \text{mBest = n0;}
12: \text{if (t < \text{maxlength}) \{}
13: \text{for (E_Iterator.connect (n0->q->arcSet); !E_Iterator.end(); E_Iterator++) \{}
14: \text{e = E_Iterator.item();}
15: \text{if (! (improvement_imposs \&\& (n0->q == e->target || better_weight (n0->rho(), e->weight, A->K)) ) \} \{
16: \text{m1 = new m (e->target, A->K->extension (n0->psi, e->weight), e, m0);
17: M1_Iterator.connect (M1);
18: m1a = M1_Iterator.search (m1, compare_function);
19: \text{if (better_n (m1, m1a, A->K, better_weight)) \{
20: M1_Iterator.replace (m1a, m1);
21: delete m1a; }
22: else
23: delete m1; } \} \} \}}
24: \text{t} ++;
25: \text{swap_M (M0, M1);}
26: \text{\}}
27: \text{return BuildPath (mBest);}
}

Fig. 2. Illustration of the similarity between pseudocode and C++ program through a modified version of the Viterbi algorithm (corresponding lines have equal numbers)
For the purpose of optimization (in the C++ program) we add a reference counter to each node $m_t$ and delete $m_t$ (and possibly some of its predecessors $m_{t-k}$) when it is no longer referenced by any successor node $m_{t+j}$. All sets $M_{t-k}$ preceding $M_t$ are deleted (without deleting all of their members $m_{t-k}$), which allows us to keep only two sets permanently, $M_t$ and $M_{t+1}$, that are swapped after each step of iteration.

4 Creating Applications with WFSC

4.1 End-User Interface

WFSC is both a compiler, creating weighted automata from different descriptions, and an interactive programming and testing environment. Easy, intuitive definition and manipulation of networks, as in Xerox’s non-weighted XFST toolkit (Karttunen et al., 1996-2003; Beesley and Karttunen, 2003), are vital to the success of an application (Aït-Mokhtar and Chanod, 1997; Grefenstette, Schiller, and Aït-Mokhtar, 2000).

A network can be described either through an enumeration of its states and transitions, including weights, or through a rational expression (i.e., a regular expression with weights).

The interactive WFSC interface provides commands for reading, writing, optimizing, exploring, visualizing, and applying networks to input. One can also create new networks from existing ones by explicitly calling operations such as union and composition. WFSC commands can be executed interactively or written to a batch file and executed as a single job. Using WFSC it is possible to read legacy non-weighted networks, created by XFST, and add weights to their states and transitions. Conversely, weights can be stripped from a weighted network to produce a non-weighted network compatible with XFST.

A new finite-state programming language is also under development (Beesley, 2003). In addition to the compilation of regular-expression and phrase-structure notations, it will provide boolean tests, imperative control structures, Unicode support, and a graphical user interface.
4.2 An Implemented Application

Optical Character Recognition (OCR) converts the bitmap of a scanned page of text into a sequence of symbols (characters) equal to the text. Post-OCR Correction attempts to reduce the number of errors in a text generated by OCR, using language models and other statistical information. This task can be performed with WFSTs (Abdallahi, 2002).

The task consists in finding the most likely corrected output text line \( \hat{o} \) in the set of all possible output lines \( O \), given an input line \( i \) generated by OCR:

\[
\hat{o} = \arg \max_{o \in O} p(o | i)
\]  

The implementation of this task with WFSC uses some basic automaton algorithms: composition, best-path search, and projection of either the input or output tape of a transducer (Figure 4):

\[
\hat{o} = \text{project}_i( \text{bestpath}( \text{project}_o( I \diamond N ) \diamond T \diamond U \diamond L ) )
\]  

First, we build a WFSA \( I \) representing the input line. Each transition of \( I \) is labeled with one (possibly incorrect) symbol of this line. Then, we construct the output-side projection of the composition of \( I \) with a WFST \( N \) representing a reverse noise model: \( \text{project}_o(I \diamond N) \). The language of the resulting WFSA contains all lines of text that could have generated the (possibly incorrect) OCR output. To find the most likely from among those lines, we compose them with a WFST \( T \), that introduces separator symbols between words, a WFST \( U \), that transforms all upper-case letters into lower-case, and a WFST \( L \), that represents a language model: \( (\ldots \diamond T \diamond U \diamond L) \). Finally, we take the input-side projection of the best path: \( \text{project}_i(\text{bestpath}(\ldots)) \).

Note that \( N \) evaluates the probability of letter sequences and \( L \) the probability of word sequences.

5 Conclusion

The article presented a new tool, WFSC, for creating, manipulating, and applying weighted finite state automata. WFSC inherits some powerful features
from Xerox’s non-weighted XFST tool, such as the “unknown symbol” and flag diacritics.

In WFSC, all algorithms work on abstract components of automata and on a generic abstract semiring, and are independent of their concrete realizations. Algorithm programmers can write in a style close to pseudocode which allows for fast prototyping. Since automaton algorithms make extensive use of set operations, special care has been given to a generic and flexible implementation of sets supporting a large number of basic operations and alternative internal structures that are inter-changeable on-the-fly.

Programmers of applications can either access WFSC’s function library through an API or create weighted automata through an end-user interface. The interface has a basic set of commands for network creation, input and output, operations on networks, network optimization, inspection, display, etc. Automata are built either from an enumeration of their states and transitions or from regular expressions that are extended to allow for specification of weights.

WFSC can be used in large-scale real-life applications. It does, however, not yet have all features initially planned. The implementation work is continuing, and due to WFSC’s generic and modular design new features and algorithms can be added easily.

Acknowledgments

We would like to thank Jean-Marc Champarnaud and Kenneth R. Beesley for their advice, and Lemine Abdallahi for his help in implementing the described application.

References

Abdallahi, Lemine. 2002. Ocr postprocessing. Internal technical report, Xerox Research Centre Europe, Meylan, France. 117


Kuich, Werner and Arto Salomaa. 1986. Semirings, Automata, Languages. Number 5 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, Germany. 110


Nicart, Florent. 2003. Toward scalable virtuality in C++. In preparation. 113


Ternary Directed Acyclic Word Graphs

Satoru Miyamoto¹, Shunsuke Inenaga¹,², Masayuki Takeda¹,², and Ayumi Shinohara¹,²

¹ Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
² PRESTO, Japan Science and Technology Corporation (JST)
{s-miya,s-ine,takeda,ayumi}@i.kyushu-u.ac.jp

Abstract. Given a set \( S \) of strings, a DFA accepting \( S \) offers a very
time-efficient solution to the pattern matching problem over \( S \). The key
is how to implement such a DFA in the trade-off between time and
space, and especially the choice of how to implement the transitions of
each state is critical. Bentley and Sedgewick proposed an effective tree
structure called ternary trees. The idea of ternary trees is to 'implant'
the process of binary search for transitions into the structure of the trees
themselves. This way the process of binary search becomes visible, and
the implementation of the trees becomes quite easy. The directed acyclic
word graph (DAWG) of a string \( w \) is the smallest DFA that accepts
all suffixes of \( w \), and requires only linear space. We apply the scheme
of ternary trees to DAWGs, introducing a new data structure named
ternary DAWGs (TDAWGs). We perform some experiments that show
the efficiency of TDAWGs, compared to DAWGs in which transitions are
implemented by tables and linked lists.

1 Introduction

Due to rapid advance in information technology and global growth of computer
networks, we can utilize a large amount of data today. In most cases, data are
stored and manipulated as strings. Therefore the development of efficient data
structures for searching strings has for decades been a particularly active research
area in computer science.

Given a set \( S \) of strings, we want some efficient data structure that enables us
to search \( S \) very quickly. Obviously a DFA that accepts \( S \) is the one. The problem
arising in implementing such an automaton is how to store the information of
the transitions in each state. The most basic idea is to use tables, with which
searching \( S \) for a given pattern \( p \) is feasible in \( O(|p|) \) time, where \( |p| \) denotes
the length of \( p \). However, the significant drawback is that the size of the tables
is proportional to the size of the alphabet \( \Sigma \) used. In particular, it is crucial
when the size of \( \Sigma \) is thousands large like in Asian languages such as Japanese,
Korean, Chinese, and so on. Using linked lists is one apparent means of escape
from this waste of memory space by tables. Although this surely reduces space
requirement, searching for pattern \( p \) takes \( O(|\Sigma| \cdot |p|) \) time in both worst and

© Springer-Verlag Berlin Heidelberg 2003
average cases. It is easy to imagine that this should be a serious disadvantage when searching texts of a large alphabet.

Bentley and Sedgewick [3] introduced an effective tree structure called ternary search trees (to be simply called ternary trees in this paper), for storing a set of strings. The idea of ternary trees is to ‘implant’ the process of binary search for transitions into the structure of the trees themselves. This way the process of binary search becomes visible, and the implementation of the trees becomes quite easy since each and every state of ternary trees has at most three transitions. Bentley and Sedgewick gave an algorithm that, for any set $S$ of strings, constructs its ternary tree in $O(|\Sigma| \cdot ||S||)$ time with $O(||S||)$ space, where $||S||$ denotes the total length of the strings in $S$. They also showed several nice applications of ternary trees [2].

We in this paper consider the most fundamental pattern matching problem on strings, the substring pattern matching problem, which is described as follows: Given a text string $w$ and pattern string $p$, examine whether or not $p$ is a substring of $w$. Clearly, a DFA that recognizes the set of all suffixes of $w$ permits us to solve this problem very quickly. The smallest DFA of this kind was introduced by Blumer et al. [4], called the directed acyclic word graph (DAWG) of string $w$, that only requires $O(|w|)$ space.

In this paper, we apply the scheme of ternary trees to DAWGs, yielding a new data structure called ternary DAWGs (TDAWGs). By the use of a TDAWG of $w$, searching text $w$ for pattern $p$ takes $O(|\Sigma| \cdot |p|)$ time in the worst case, but the time complexity for the average case is $O(|\Sigma| \cdot |p|)$, which is an advantage over DAWGs implemented with linked lists that require $O(|\Sigma| \cdot |p|)$ expected time. Therefore, the key is how to construct TDAWGs quickly. Note that the set of all suffixes of a string $w$ is of size quadratic in $|w|$. Namely, simply applying the algorithm by Bentley and Sedgewick [3] merely allows us to construct a TDAWG of $w$ in $O(|\Sigma| \cdot |w|^2)$ time. However, using a modification of the on-line algorithm of Blumer et al. [4], pleasingly, the TDAWG of $w$ can be constructed in $O(|\Sigma| \cdot |w|)$ time. We also performed some computational experiments to evaluate the efficiency of TDAWGs, using English texts, by the comparison with DAWGs implemented by tables and linked lists. The most interesting result is that the construction time of TDAWGs is dramatically faster than those of DAWGs with tables and linked lists. Plus, it is evaluated that search time by TDAWGs is also faster than that by DAWGs with linked lists.

## 2 Directed Acyclic Word Graphs

Let $\Sigma$ be a finite alphabet. An element of $\Sigma^*$ is called a string. Strings $x$, $y$, and $z$ are said to be a prefix, substring, and suffix of string $w = xyz$, respectively. The sets of prefixes, substrings, and suffixes of a string $w$ are denoted by $\text{Prefix}(w)$, $\text{Substr}(w)$, and $\text{Suffix}(w)$, respectively. The length of a string $w$ is denoted by $|w|$. The empty string is denoted by $\varepsilon$, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Let $S \subseteq \Sigma^*$. The number of strings in $S$ is denoted by $|S|$, and the sum of the lengths of strings in $S$ by $||S||$. 
The following problem is the most fundamental and important in string processing.

**Definition 1 (Substring Pattern Matching Problem).**

**Instance:** a text string $w \in \Sigma^*$ and pattern string $p \in \Sigma^*$.

**Determine:** whether $p$ is a substring of $w$.

It is clear that this problem is solvable in time proportional to the length of $p$, by using an automaton that accepts $\text{Substr}(w)$. The most basic automaton of this kind is the suffix trie. The suffix trie of a string $w \in \Sigma^*$ is denoted by $\text{STrie}(w)$. What is obtained by minimizing $\text{STrie}(w)$ is called the directed acyclic word graph (DAWG) of $w$ [9], denoted by $\text{DAWG}(w)$. In Fig. 1 we show $\text{STrie}(w)$ and $\text{DAWG}(w)$ with $w = \text{cocoa}$.

The initial state of $\text{DAWG}(w)$ is also called the source state, and the state accepting $w$ is called the sink state of $\text{DAWG}(w)$. Each state of $\text{DAWG}(w)$ other than the source state has a suffix link. Assume $x_1, \ldots, x_k$ are the substrings of $w$ accepted in one state of $\text{DAWG}(w)$, arranged in the decreasing order of their lengths. Let $y a = x_k$, where $y \in \Sigma^*$ and $a \in \Sigma$. Then the suffix link of the state accepting $x_1, \ldots, x_k$ points to the state in which $y$ is accepted.

DAWGs were first introduced by Blumer et al. [4], and have widely been used for solving the substring pattern matching problem, and in various applications [7, 8, 15].

**Theorem 1 (Crochemore [6]).** For any string $w \in \Sigma^*$, $\text{DAWG}(w)$ is the smallest (partial) DFA that recognizes $\text{Suffix}(w)$.

**Theorem 2 (Blumer et al. [4]).** For any string $w \in \Sigma^*$ with $|w| > 1$, $\text{DAWG}(w)$ has at most $2|w| - 1$ states and $3|w| - 3$ transitions.

It is a trivial fact that $\text{DAWG}(w)$ can be constructed in time proportional to the number of transitions in $\text{STrie}(w)$ by the DAG-minimization algorithm.
Fig. 2. The on-line construction of $DAWG(w)$ with $w = \text{cocoao}$. The solid arrows are the transitions, and the dashed arrows are the suffix links. Note that the state pointed by the suffix link of the sink state will be the active state of the next phase. In the process of updating $DAWG(\text{cocoa})$ to $DAWG(\text{cocoao})$, the state accepting $\{\text{co}, \text{o}\}$ is separated into two states for $\{\text{co}\}$ and $\{\text{o}\}$ by Revuz [13]. However, the number of transitions of $STrie(w)$ is unfortunately quadratic in $|w|$. The direct construction of $DAWG(w)$ in linear time is therefore significant, in order to avoid creating redundant states and transitions that are deleted in the process of minimizing $STrie(w)$. Blumer et al. [4] indeed presented an algorithm that directly constructs $DAWG(w)$ and runs in linear time if $\Sigma$ is fixed, by means of suffix links. Their algorithm is on-line, namely, for any $w \in \Sigma^*$ and $a \in \Sigma$ it allows us to update $DAWG(w)$ to $DAWG(\text{wa})$ in amortized constant time, meaning that we need not construct $DAWG(\text{wa})$ from scratch.

We here briefly recall the on-line algorithm by Blumer et al. It updates $DAWG(w)$ to $DAWG(\text{wa})$ by inserting suffixes of $\text{wa}$ into $DAWG(w)$ in decreasing order of their lengths. Let $z$ be the longest string in $\text{Substr}(w) \cap \text{Suffix}(\text{wa})$. 
Then \( z \) is called the longest repeated suffix of \( wa \) and denoted by \( LRS(wa) \). Let \( z' = LRS(w) \). Let \( |wa| = l \) and \( u_1, u_2, \ldots, u_t, u_{t+1} \) be the suffixes of \( wa \) ordered in their lengths, that is, \( u_1 = wa \) and \( u_{t+1} = \varepsilon \). We categorize these suffixes of \( wa \) into the following three groups.

**Group 1** \( u_1, \ldots, u_{i-1} \)

**Group 2** \( u_i, \ldots, u_{j-1} \) where \( u_i = z'a \)

**Group 3** \( u_j, \ldots, u_{l+1} \) where \( u_j = z \)

Note all suffixes in Group 3 are already represented in \( DAWG(w) \). We can insert all the suffixes of Group 1 into \( DAWG(w) \) by creating a new transition labeled by \( a \) from the current sink state to the new sink state. It obviously takes only constant time. Therefore, we have only to care about those in Group 2. Let \( v_i, \ldots, v_{j-1} \) be the suffixes of \( w \) such that, for any \( i \leq k \leq j - 1 \), \( v_k a = u_k \). We start from the state corresponding to \( LRS(w) = z' = v_i \) in \( DAWG(w) \), which is called the active state of the current phase. A new transition labeled by \( a \) is inserted from the active state to the new sink state. The state to be the next active state is found simply by traversing the suffix link of the state for \( v_i \), in constant time, and a new transition labeled by \( a \) is created from the new active state to the sink state. After we insert all the suffixes of Group 2 this way, the automaton represents all the suffixes of \( wa \).

We now pay attention to \( LRS(wa) = z = u_j \). Let \( x \) be the longest string in the state where \( u_j \) is accepted. We then have to check whether \( x = u_j \) or not. If not, the state is separated into two states, where one accepts the longer strings than \( u_j \), and the other accepts the rest. Associating each state with the length of the longest string accepted in it, we can deal with this state separation in constant time.

The on-line construction of \( DAWG(cocoa) \) is shown in Fig. 2.

### 3 Ternary Directed Acyclic Word Graphs

Bentley and Sedgewick \[3, 2\] introduced a new data structure called ternary trees, which are quite useful for storing a set of strings, from both viewpoints of space efficiency and search speed. The idea of ternary trees is to ‘implant’ the process of binary search for linked lists into the trees themselves. This way the process of binary search becomes visible, and the implementation of the trees becomes quite easy since each and every state of ternary trees has at most three transitions.

The left figure in Fig. 3 is a ternary tree for \( Suffix(w) \) with \( w = cocoa \). We can see that this corresponds to \( STrie(w) \) in Fig. 1, and therefore, the tree is called a ternary suffix trie (TSTrie) of string \( cocoa \).

For a substring \( x \) of a string \( w \in \Sigma^* \), we consider set \( CharSet_w(x) = \{ a \in \Sigma \mid xa \in Substr(w) \} \) of characters. In \( STrie(w) \), each character of \( CharSet_w(x) \) is associated with a transition from state \( x \) (see \( STrie(cocoa) \) in Fig. 1). However, in a TSTrie of \( w \), each character in \( CharSet_w(x) \) corresponds to a state. This means that we can regard \( CharSet_w(x) \) as a set of the states that immediately
follows string $x$ in the TSTrie of $w$, where elements of $CharSet_w(x)$ are arranged in lexicographical order, top-down. There are many variations of the arrangement of elements in $CharSet_w(x)$, but we arrange them in increasing order of their leftmost occurrences in $w$, top-down. Thus the arrangement of the states is uniquely determined, and the resulting structure is called the TSTrie of $w$, denoted by $TSTrie(w)$. The state corresponding to the character in $CharSet_w(x)$ with the earliest occurrence, is called the top state with respect to $CharSet_w(x)$, since it is arranged on the top of the states for characters in $CharSet_w(x)$.

Given a pattern $p$, at any node of $TSTrie(w)$ we examine if the character $a$ in $p$ we currently focus on is lexicographically larger than the character $b$ stored in the state. If $a < b$, then we take the left transition from the state and compare $a$ to the character in the next state. If $a > b$, then we take the right transition from the state and compare $a$ to the character in the next state. If $a = b$, then we take the center transition from the state, now the character $a$ is recognized, and we compare the next character in $p$ to the character in the next state. We give a concrete example of searching for pattern $oa$ using $TSTrie(cocoa)$ in Fig. 3.

We start from the initial state of the tree and have $o > c$, and thus go down to the next state via the right transition. At the next state we have $o = o$, and thus we take the center transition from the state and arrive at the next state, with the character $o$ recognized. We then compare the next character $a$ in the pattern with $c$ in the state where we are. Now we have $a < c$, we go down along the left transition of the state and arrive at the next state, where we have $a = a$.

Then we take the center transition and arrive at the next state, where finally $oa$ is accepted. This way, for any pattern $p \in \Sigma^*$ we can solve the substring pattern matching problem of Definition 1 in $O(|\Sigma| \cdot |p|)$ expected time.

We now consider to apply the above scheme to $DAWG(w)$. What is obtained here is the ternary $DAWG$ ($TDAWG$) of $w$, denoted by $TDAWG(w)$. The right figure in Fig. 3 is $TDAWG(cocoa)$. Compare it to $DAWG(cocoa)$ in Fig. 1 and $TSTrie(cocoa)$ in Fig. 3. It is quite obvious that using $TDAWG(w)$ we can examine if $p \in Substr(w)$ in $O(|\Sigma| \cdot |p|)$ expected time, as well. Reasonably,
The on-line construction of $TDAWG(w)$ with $w = \text{cocoa}$. The dashed arrows are the suffix links. Notice only top states have suffix links, and are pointed by suffix links of other top states. In the process of updating $TDAWG(\text{cocoa})$ to $TDAWG(\text{cocoao})$, the state accepting $\{\text{co, o}\}$ is separated into two states for $\{\text{co}\}$ and $\{\text{o}\}$, as well as the case of DAWGs shown in Fig. 2.

$TDAWG(w)$ can be constructed by the on-line algorithm of Blumer et al. [4] that was recalled in Section 2. In $TDAWG(w)$ only top states have suffix links, and can be directed by suffix links of other states.

The on-line construction of $TDAWG(\text{cocoa})$ is shown in Fig. 4.

**Theorem 3.** For any string $w \in \Sigma^*$, $TDAWG(w)$ can be constructed on-line, in $O(|\Sigma| \cdot |w|)$ time using $O(|w|)$ space.

### 4 Experiments

In this section we show some experimental results that reveal the advantage of our TDAWGs, compared to DAWGs with tables (table_DAWGs) and DAWGs with linked lists (list_DAWGs). The tables were implemented by arrays of length 256. The linked lists were linearly searched at any state of the
Fig. 5. The upper left chart is the memory requirements (in Mbytes) of TDAWGs and list_DAWGs. The upper right chart is the construction times (in seconds) of TDAWGs, list_DAWGs and table_DAWGs. The lower chart is the searching time (in micro seconds) of TDAWGs, list_DAWGs and table_DAWGs.

list_DAWGs. All the three algorithms to construct TDAWGs, table_DAWGs and list_DAWGs were implemented in the C language. All calculations were performed on a Laptop PC with PentiumIII-650MHz CPU and 256MB main memory running VineLinux2.6r2. We used the English text “ohsumed.91” available at http://trec.nist.gov/data.html.

The upper left chart of Fig. 5 shows memory requirements for TDAWGs and list_DAWGs, where memory spaces for both grow linearly, as expected. One can see that TDAWGs require about 20% more memory than list_DAWGs. The memory requirement of table_DAWGs is not shown since it is too much for the scale of the chart. The table_DAWG for the text of size 64KB required 98.82MB of memory space, and that for the text of size 128KB needed 197.53MB. Thus table_DAWGs are rather unusable in reality.
The second test was the construction times for TDAWGs, table_DAWGs, and list_DAWGs, shown upper right of Fig. 5. One can see that the TDAWGs were constructed about twice faster than the list_DAWGs. This seems the effect of binary search in the TDAWGs, while the linked lists were linearly searched in the list_DAWGs. As for the table_DAWGs, though searching the transition can be done in constant time, the memory allocation for the tables seemed to take too much time.

The third test was searching times for patterns of different lengths. We randomly chose 100 substrings of the text for each length, and searched for every of them 1 million times. The result shown in the lower chart in Fig. 5 is the average time of searching for a pattern once. Remark that the TDAWGs are faster than list_DAWGs, even for longer patterns.

5 Conclusions and Further Work

In this paper we introduced a new data structure called ternary directed acyclic word graphs (TDAWGs). The process of binary search for the transitions is ‘implanted’ in each state of TDAWGs. For any string \( w \in \Sigma^* \), \( TDAWG(w) \) can be constructed in \( O(|\Sigma| \cdot |w|) \) time using \( O(|w|) \) space, in on-line fashion. Our experiments showed that TDAWGs can be constructed much faster than both DAWGs with tables and DAWGs with linked lists, for English texts. Moreover, searching time of TDAWGs is also better than that of DAWGs with linked lists. Thinking over the fact that TDAWGs are better in speed than the two other types of DAWGs though the alphabet size of the English text is only 256, TDAWGs should be a lot more effective when applied to texts of large alphabet such as Japanese, Korean, Chinese, and so on. We emphasize that the benefit of the ternary-based implementation is not limited to DAWGs. Namely, it can be applied to any automata-oriented index structure such as suffix trees \([16, 12, 14, 10]\) and compact directed acyclic word graphs (CDAWGs) \([5, 9, 11]\). Therefore, we can also consider ternary suffix trees and ternary CDAWGs. Concerning the experimental results on TDAWGs, ternary suffix trees and ternary CDAWGs are promising to perform very well in practice.

As previously mentioned, the search time for pattern \( p \) using DAWGs with linked lists is \( O(|\Sigma| \cdot |p|) \) in the worst case, but there is a way to improve it to \( O(\log |\Sigma| \cdot |p|) \) time with additional effort for performing binary search. However, it is not practical since the additional work consumes a considerable amount of time, and is not implementation-friendly. Further, it does not allow us to update the input text string since it is just an off-line algorithm. However, as for TDAWGs, we can apply the technique of AVL trees \([1]\) for balancing the states of TDAWGs. In this scheme, the states for \( CharSet_w(x) \) for any substring \( x \) are “AVL-balanced”, and thus the time to search for pattern \( p \) is \( O(\log |\Sigma| \cdot |p|) \) even in the worst case. The difference from DAWGs with linked lists is that we can construct “AVL-balanced” TDAWGs in on-line manner, directly, and can update the input text string easily. Moreover, the construction algorithm runs in \( O(\log |\Sigma| \cdot |w|) \) time unlike the case of linked lists requiring \( O(|\Sigma| \cdot |w|) \) time.
Therefore, TDAWGs are more practical and flexible for guaranteeing $O(\log |\Sigma| \cdot |p|)$-time search in the worst case.

Moreover, there is a variation of TDAWGs that is more space-economical. Note that Fig. 3 of Section 3 can be minimized by the algorithm of Revuz [13], and the resulting structure is shown in Fig. 6, which is called the minimum TDAWG (MTDAWG) of the string. To use Revuz’s algorithm we have to maintain the reversed transition for every transition, and it for sure requires too much space. Thus we are now interested in an on-line algorithm to construct MTDAWGs directly, but it is still incomplete. We expect that searching for pattern strings using MTDAWGs will be faster than using TDAWGs, since memory allocation for MTDAWGs is likely to be more quick.

References


Running Time Complexity
of Printing an Acyclic Automaton

Franck Guingne\textsuperscript{1,2}, André Kempe\textsuperscript{1}, and Florent Nicart\textsuperscript{1,2}

\textsuperscript{1} Xerox Research Centre Europe, Grenoble Laboratory
6 chemin de Maupertuis, 38240 Meylan, France
Firstname.Lastname@xrce.xerox.com
http://www.xrce.xerox.com

\textsuperscript{2} Laboratoire d’Informatique Fondamentale et Appliquée de Rouen
Faculté des Sciences et des Techniques, Université de Rouen
76821 Mont-Saint-Aignan, France
firstname.lastname@dir.univ-rouen.fr
http://www.univ-rouen.fr/LIFAR/

Abstract. This article estimates the worst-case running time complexity for traversing and printing all successful paths of a normalized trim acyclic automaton. First, we show that the worst-case structure is a festoon with distribution of arcs on states as uniform as possible. Then, we prove that the complexity is maximum when we have a distribution of $e$ (Napier constant) outgoing arcs per state on average, and that it can be exponential in the number of arcs.

1 Introduction

This article takes place in the scope of the study of complexity of automata algorithms (Yu, Zhuang, and Salomaa, 1994), and particularly in the study of the worst-case complexity (Nicaud, 2000). We estimate the worst-case running time complexity for traversing and printing all successful paths of a finite-state automaton. The number of states and arcs are given, and the structure is unknown. This task occurs, e.g., when all words of a natural-language lexicon represented through an automaton are printed into a file.

By “printing a path” we mean that the label of the path is written out when its final state is reached. Hence, the complexity of this part of the task depends on both the number and the length of all paths. The required traversal of paths has in general a much lower complexity because prefixes shared by several paths are traversed only once, so that many paths are not traversed in full length.

We start from the worst-case structure and show that any other structure decreases the complexity of the task. We restrict our analysis to acyclic automata with a single initial state and a single final state.

The article is structured as follows: Section 2.1 recalls some basic notions concerning automata and lists the assumptions made for all following estimations. Section 3 shows the automaton structure that maximizes the analyzed
complexity. Section 4 and 5 estimate the complexity for different cases, and Section 6 reports some numerical calculations, w.r.t. the number of arcs. Section 7 presents our outcomes in a concise form and concludes the article.

2 Preliminaries

2.1 Automata

An automaton $A$ is defined by the 5-tuple $\langle \Sigma, Q, I, F, E \rangle$ (Eilenberg, 1974) (Hopcroft, Motwani, and Ullman, 2001) where

- $\Sigma$ is the finite alphabet
- $Q$ is the finite set of states
- $I \subseteq Q$ is the set of initial states
- $F \subseteq Q$ is the set of final states
- $E \subseteq Q \times \Sigma \times Q$ is the finite set of arcs

An automaton is said to be normalized if and only if it has exactly one initial state with no incoming arc and one final state with no outgoing arc (Berstel, 1989).

A state $s$ is reachable (resp. coreachable) if there exists a path from some state of $I$ to $s$ (resp. a path from $s$ to some state of $F$); an automaton is said to be trim if and only if all its states are reachable and coreachable (Perrin, 1990).

For any arc $e \in E$ we denote by

- $p(e) : E \rightarrow Q$ the source state of $e$
- $n(e) : E \rightarrow Q$ the target state of $e$

Symbols are required only for printing out the paths. They are irrelevant in the estimation of the complexity since the complexity for printing out a path does not depend on the symbols themselves.

A path $\pi$ of length $l = |\pi|$ is a sequence of arcs $e_1 e_2 \cdots e_l$ such that $n(e_i) = p(e_{i+1})$ for all $i \in [1, l-1]$. A path is said to be successful if and only if $p(e_1) \in I$ and $n(e_l) \in F$. The set of all successful paths of $A$ is denoted by $\Pi$.

2.2 Conventions and Assumptions

To simplify our notation, we will denote by:

- $a = |E|$ the number of arcs in $A$
- $s = |Q|$ the number of states in $A$
- $p = |\Pi|$ the number of successful paths in $A$

The following analysis is made for an automaton $A$ under the assumption that:

- $A$ is acyclic
- $A$ is normalized
- $A$ is trim
- $a$ and $s$ are given

No more assumptions are needed for our study. Our outcomes are independent of possible additional properties of the automaton, such as determinism, $\varepsilon$-arcs, multiplicities (or weights).
3 Worst-Case Structure

This section introduces the structure that maximizes the complexity of traversing and printing all successful paths of a normalized acyclic automaton. We start from the worst-case structure and show that any other structure decreases the complexity.

Let $A$ be an automaton that satisfies the assumptions in section 2.2 and has the structure shown in Figure 1, with $a$ arcs, $s$ states, and $p$ paths of length $l$. Every state $q_i$, except the last (final) one has $n_i$ outgoing arcs leading to the next state $q_{i+1}$.

Only $a$ and $l$ are fixed. According to the above structure, $s = l + 1$. We will, however, discuss alternative structures in the case of splitting a state (Figure 2) where $s > l + 1$ ($s$ will be temporarily variable).

Since the analyzed complexity depends on both $p$ and $l$, and since $l$ is fixed at present, the maximum of the complexity is reached with the maximum of $p$. Hence we will maximize $p$ in this section.

**Proposition 1.** Let us consider a structure as shown in Figure 1. Let $n = \lfloor \frac{a}{l} \rfloor$, and let $q_i$, for $i \in [0, l-1]$, such that

$$\text{out}(q_i) = \begin{cases} n & \text{if } i < l - (a \mod l) \\ n + 1 & \text{otherwise.} \end{cases}$$

(1)

Then the maximum number of paths is:

$$P_{\text{max}} = \prod_{i=0}^{l-1} \text{out}(q_i)$$

(2)

When $a \mid l$ ($l$ divides $a$) we denote

$$P_{\text{max}} = p_{\text{uni}} = n^l$$

(3)
With respect to the notion of hammock used in (Caron and Ziadi, 2000) and (Giammarresi, Ponty, and Wood, 2001), the structure defined by Proposition 1 is a uniform acyclic multi-hammock. In the following we call it a festoon.

**Proof.** Any of the following changes to this structure will reduce the number of paths.

1. **Moving Arcs to Other States:** if one arc is moved from $q_i$ to $q_j$, so that $q_i$ will have $n - 1$, $q_j$ will have $n + 1$, and all the other states will have $n$ arcs, then the number of paths will decrease to:

   $$p_1 = n^{l-2} (n - 1) (n + 1) = n^{l-2} (n^2 - 1) = n^l - n^{l-2}$$

   $$= p_{uni} - n^{l-2} \quad (4)$$

   If $k$ arcs are moved in that way, the number of paths decreases as well:

   $$p_k = n^{l-2} (n - k) (n + k) = n^{l-2} (n^2 - k^2)$$

   $$= p_{uni} - k^2 n^{l-2} \quad (5)$$

   If uniform distribution is impossible because $q \notin \mathbb{N}$ then the maximum number of paths is reached when the distribution of the arcs is given by the function $out(q)$ i.e., $n$ or $n+1$ arcs per state. For a length $l = l_1 + l_2$ with $l_1$ states having $n$ arcs each and $l_2$ states having $n+1$ arcs each, the number of path is:

   $$P_{max} = n^{l_1} \cdot (n + 1)^{l_2} \quad (6)$$

   If we move an arc from an $n+1$-arcs to an $n$-arcs section then obviously the number of paths does not change. However, if we move an arc from an $n+1$-arcs to another $n+1$-arcs section, then the number of path decreases to:

   $$p = n^{l_1} \cdot (n + 1)^{l_2-2} n(n + 2) = n^{l_1} \cdot (n + 1)^{l_2-2} \cdot ((n + 1)^2 - 1)$$

   $$= n^{l_1} \cdot (n + 1)^{l_2-2} - n^{l_1} (n + 1)^{l_2-2}$$

   $$= P_{max} - n^{l_1} (n + 1)^{l_2-2} \quad (7)$$

   and if we move an arc from an $n$-arcs to another $n$-arcs section, it decreases (symmetrically) to:

   $$p = P_{max} - (n + 1)^{l_2} n^{l_1-2} \quad (8)$$

   Any other move of $k$ arcs between any two sections can be obtained by combining the listed moves.

2. **Splitting of States:** If there are $\varphi$ prefixes ending and $\sigma$ suffixes starting in a state $q$ then the number of paths traversing $q$ is (Figure 2a):

   $$\dot{p} = \varphi \cdot \sigma \quad (9)$$
If we split \( q \) and its sets of prefixes and suffixes, so that there will be two new states, \( q_1 \) with \( \varrho_1 \) prefixes and \( \sigma_1 \) suffixes, and \( q_2 \) with \( \varrho_2 \) prefixes and \( \sigma_2 \) suffixes, such that \( \varrho_1 + \varrho_2 = \varrho \) and \( \sigma_1 + \sigma_2 = \sigma \), then the number of paths traversing either \( q_1 \) or \( q_2 \) is reduced to (Figure 2b):

\[
\hat{p}_1 + \hat{p}_2 = \varrho_1 \cdot \sigma_1 + \varrho_2 \cdot \sigma_2 = (\varrho - \varrho_2) \cdot \sigma_1 + (\varrho - \varrho_1) \cdot \sigma_2 = \varrho \cdot (\sigma_1 + \sigma_2) - (\varrho_2 \cdot \sigma_1 + \varrho_1 \cdot \sigma_2)
\]

\[
\varrho_2 \cdot \sigma_1 + \varrho_1 \cdot \sigma_2 > 0 \Rightarrow \hat{p}_1 + \hat{p}_2 < \hat{p}
\]

Splitting a state \( q \) into \( k \) states \( q_1 \) to \( q_k \) has the same effect (Figure 2c):

\[
\sum_{i=1}^{k} \hat{p}_i = \varrho \cdot \sigma - \sum_{i=1}^{k} \left( \sum_{j=1, j \neq i}^{k} \varrho_j \right) \cdot \sigma_i
\]

with \( \varrho = \sum_{i=1}^{k} \varrho_i \) and \( \varrho = \sum_{i=1}^{k} \varrho_i \)

3. Changing the Source or Destination of Arcs:

The number of suffixes that follow an arc is depending on the length of the suffixes (Figure 1). If an arc leading from \( q_i \) to \( q_{i+1} \) is redirected to a following destination \( q_{i+m} \) \((m \geq 2)\) then the number of suffixes following that arc decreases and hence the total number of suffixes decreases too (Figure 3). Redirecting the arc to a preceding destination state \( q_{i-b} \) \((b \geq 0)\) would make the automaton cyclic and is therefore not in the scope of this investigation.

Changing the source state of an arc will lead to similar results as changing its destination.

For an acyclic automaton with a given number of arcs, \( a \), and a fixed length, \( l \), the maximal number of paths, \( P_{max} \), is reached with the festoon structure as in Figure 1 and with an as uniform as possible distribution of \( a \) arcs per state on
average (except for the final state). This is because any other structure can be obtained from this one by combining the three modifications above, which all reduce $P_{max}$.

Since this structure maximizes $p$ for any fixed $l$, it does it also for variable $l$.

4 Worst-Case Complexity for Variable Length

Let us consider a festoon, with fixed $a$, and variable $s$ and $l$. The number $n$ of arcs per state given by the function $out(q)$ will depend on $l$. However, different $l$ will lead to different $p$.

The complexity of traversing and printing all paths of $A$, depends on the number of arcs to be “handled” (i.e., traversed or printed). This number is given by the following function:

$$f(n) = k p l$$

with $k \in [1, 2]$ \quad $p = n^l$ \quad $l = \frac{a}{n}$

The coefficient $k \in [1, 2]$ expresses that each of the $l$ arcs on each of the $p$ paths is handled either once (only printed because already traversed on another path) or twice (traversed and printed). Although $k$ is depending on $n$ and $l$, we will consider it as a constant. It has no effect on the complexity of the current task.

To find the number $\hat{n}$ of arcs per state that leads to the worst-case complexity, we compute the real $\hat{x}$ which maximizes $f(x) = k l x^l$. We construct the first derivative of $f(x)$:

$$f(x) = k x^l l = k x^{ax - 1} a x^{-1} = k a x^{ax - 1 - 1} = k a e^{(ax - 1 - 1)\ln x}$$

$$f'(x) = k a e^{\psi}$$

with $\psi = (ax - 1 - 1)\ln x$

$$f'(x) = k a \psi e^{\psi}$$

$$f'(x) = k a \phi e^{\psi}$$

with $\phi = \psi' = a x^{-2}(1 - \ln x) - x^{-1}$
To find all extrema, we equate the first derivative to 0. Since for all \( k, a, \psi \) we have: \( k a e^{\psi} > 0 \), we get:

\[
 f'(x=\hat{x}) = 0 \quad \Rightarrow \quad \phi(x=\hat{x}) = 0 \\
 0 = a \hat{x}^{-2}(1 - \ln \hat{x}) - \hat{x}^{-1} \\
 \frac{1}{a} = 1 - \ln \hat{x} \tag{16}
\]

For an automaton with just one arc, \( \hat{x} \) is obviously 1:

\[
 \frac{1}{1} = 1 - \ln 1
\]

For large automata Equation 16 means:

\[
 \lim_{a \to \infty} \frac{1}{a} = 0 \quad \Rightarrow \quad 0 = 1 - \ln \hat{x} \\hat{x} = e \tag{17}
\]

We further analyze \( f'(x) \) to see for which values \( x = \hat{x} \) the function \( f(x) \) is growing:

\[
 f'(x=\hat{x}) > 0 \quad \Rightarrow \quad \phi(x=\hat{x}) > 0 \\
 0 < a \hat{x}^{-2}(1 - \ln \hat{x}) - \hat{x}^{-1} \\
 \frac{1}{a} < 1 - \ln \hat{x} \hat{x} < e \tag{18}
\]

and for which values \( x = \hat{x} \) the function \( f(x) \) is falling:

\[
 f'(x=\hat{x}) < 0 \quad \Rightarrow \quad \phi(x=\hat{x}) < 0 \\
 0 > a \hat{x}^{-2}(1 - \ln \hat{x}) - \hat{x}^{-1} \\
 \frac{1}{a} > 1 - \ln \hat{x} \hat{x} > e \tag{19}
\]

Equations 17, 19, and 21 show that \( f(x) \) has its only maximum at \( x = e \), is monotonically ascending for all \( x < e \), and monotonically descending for all \( x > e \). The maximal number of arcs to be handled is:

\[
 f(x=\hat{x}=e) = k a e^{a e^{-1} - 1} = \frac{k}{e} a e^{a} \tag{22}
\]

Hence, the worst-case complexity of the above task, with fixed \( a \) and variable \( s \) and \( l \) is:

\[
 \mathcal{O}( f(x) ) = \mathcal{O}( e^{a} ) = \mathcal{O}( 1.4447^{a} ) \tag{23}
\]
5 Worst Case for a Given Number of States

In the previous two sections we made no assumption on $s$ and $l$. In the present section $s$ will be fixed, and $l$ will be variable and ignored in the remainder of our analysis. This corresponds with our initial assumptions (Section 2.2).

Let $A$ now be an automaton with fixed $a$ and $s$. In this case the results of Section 3 and Section 4 may seem contradictory: $s$ seems to impose $l = s - 1$, and $s$ and $a$ together seem to impose $x = \frac{a}{s}$, the number of arcs per state. This leads us to the question whether for $\frac{a}{s} \neq e$, the worst-case complexity is reached with $x = e$ or with uniform distribution $x = \frac{a}{s}$. In fact there is no contradiction, but we have to distinguish two different cases:

1. If the number of states, $s$, is below the limit $s < \frac{a}{e} + 1$ then the worst case is reached with a structure as in Figure 1 and $l = s - 1$:

   $$f(x) = k \ p \ l = k \ x \ l = k \ \left( \frac{a}{l} \right)^{l} l$$

   $$= k \ \left( \frac{a}{s-1} \right)^{s-1} (s-1)$$  \hspace{1cm} (24)

   This agrees with both previous results: the arcs are (approximately) uniformly distributed with $x = \frac{a}{l} > e$. To further increase the complexity, $x$ would have to decrease towards $e$, which is not possible because it would require more than $a$ states. The complexity of this case is:

   $$\mathcal{O} \left( f(x) \left| s - 1 < \frac{a}{e} \right. \right) = \mathcal{O} \left( \left( \frac{a}{s-1} \right)^{s-1} \right)$$  \hspace{1cm} (25)

2. If the number of states, $s$, exceeds the limit $s > \frac{a}{e} + 1$ then the worst case is reached with a length $l = \frac{a}{e}$, using $l + 1$ states to form a structure as in Figure 1, and the remaining states on state-splitting as in Figure 2 (Section 3, Point 2). This splitting will decrease the complexity, so that Equation 23 constitutes an (unreached) upper bound in this case:

   $$\mathcal{O} \left( f(x) \left| s - 1 > \frac{a}{e} \right. \right) < \mathcal{O} \left( \sqrt{e}^{a} \right) = \mathcal{O} (1.4447^{a})$$  \hspace{1cm} (26)

   This agrees with both previous conclusions: the arcs are (approximately) uniformly distributed with $x = \frac{a}{l}$, and $x$ equals to $e$, the value that maximizes the complexity.

   If the number of states $s$ equals the limit $s = \frac{a}{e} + 1$ then both previous cases hold and the equations of their complexities (25, 26) provide the same value.
6 Complexity Calculations for Some Cases

Table 1 shows results from a calculation of the function $f_a(x) = kpl$ (Equation 13), describing the task of traversing and printing a normalized acyclic automaton, given the worst-case structure, fixed $a$, and variable $s$ and $l$. The coefficient $k$ is set to 2. Each row gives (for fixed $a$) the average number of arcs per state, $\hat{x}$, the length, $\hat{l}$, and the number of paths, $\hat{p}$, where $f_a(x)$ reaches its maximum:

$$k = 2$$
$$\hat{x} = \arg\max_x f_a(x)$$
$$\hat{l} = \frac{a}{\hat{x}}$$
$$\hat{p} = \hat{x}^{\hat{l}}$$
$$f_a(\hat{x}) = k \hat{p} \hat{l} = \max_x f_a(x)$$

according to Equation 13. Note that $\hat{x} \in \mathbb{R}$ because it is an average over all states, and that in fact all $\hat{l}, \hat{p} \in \mathbb{N}$ rather than $\hat{l}, \hat{p} \in \mathbb{R}$. Thus the table gives an approximation in $\mathbb{R}$ of values that are actually in $\mathbb{N}$.

For example, in an automaton with 16 arcs ($a = 16$), the maximum is reached in fact with $l = 7$ ($\hat{l} = 6.82$), $x' = \hat{x} = \frac{a}{\hat{l}} = 2.285714$, ($\hat{x} = 2.3474$, $x_i \in \{2, 3\}$), $p = 2^5 \cdot 3^2 = 288$ ($\hat{p} = 335.7$), and $f_a(x') = 4032$ ($f_a(\hat{x}) = 4576$). With growing $a$, $\hat{x}$ approaches $e = 2.718282 \ldots$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$\hat{x}$</th>
<th>$\hat{l}$</th>
<th>$\hat{p}$</th>
<th>$f_a(\hat{x})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0000</td>
<td>1.000</td>
<td>1.000</td>
<td>2.000</td>
</tr>
<tr>
<td>2</td>
<td>1.3702</td>
<td>1.460</td>
<td>1.584</td>
<td>4.623</td>
</tr>
<tr>
<td>4</td>
<td>1.7535</td>
<td>2.280</td>
<td>3.601</td>
<td>16.43</td>
</tr>
<tr>
<td>8</td>
<td>2.0926</td>
<td>3.820</td>
<td>16.830</td>
<td>128.7</td>
</tr>
<tr>
<td>16</td>
<td>2.3474</td>
<td>6.827</td>
<td>335.70</td>
<td>4576</td>
</tr>
<tr>
<td>32</td>
<td>2.5130</td>
<td>12.730</td>
<td>1.247 \times 10^5</td>
<td>3.177 \times 10^6</td>
</tr>
<tr>
<td>64</td>
<td>2.6096</td>
<td>24.520</td>
<td>1.646 \times 10^7</td>
<td>8.073 \times 10^{11}</td>
</tr>
<tr>
<td>128</td>
<td>2.6623</td>
<td>48.080</td>
<td>2.791 \times 10^9</td>
<td>2.684 \times 10^{22}</td>
</tr>
<tr>
<td>256</td>
<td>2.6898</td>
<td>95.170</td>
<td>7.913 \times 10^{10}</td>
<td>1.506 \times 10^{43}</td>
</tr>
<tr>
<td>512</td>
<td>2.7040</td>
<td>189.350</td>
<td>3.311 \times 10^{11}</td>
<td>2.390 \times 10^{44}</td>
</tr>
<tr>
<td>1024</td>
<td>2.7112</td>
<td>377.700</td>
<td>3.998 \times 10^{12}</td>
<td>3.020 \times 10^{46}</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
<td></td>
<td>program numeric overflow</td>
</tr>
</tbody>
</table>

Table 1. Calculation of the worst-case complexity with fixed $a$ and variable $s$ and $l$. 

For example, in an automaton with 16 arcs ($a = 16$), the maximum is reached in fact with $l = 7$ ($\hat{l} = 6.82$), $x' = \hat{x} = \frac{a}{\hat{l}} = 2.285714$, ($\hat{x} = 2.3474$, $x_i \in \{2, 3\}$), $p = 2^5 \cdot 3^2 = 288$ ($\hat{p} = 335.7$), and $f_a(x') = 4032$ ($f_a(\hat{x}) = 4576$). With growing $a$, $\hat{x}$ approaches $e = 2.718282 \ldots$
7 Conclusion

Our investigation has shown (Equations 25, 26) that the complexity of traversing and printing all paths of a normalized acyclic automaton with \( s \) states and \( a \) arcs reaches its maximum with a structure as in Figure 1 and (approximately) uniform distribution of arcs over the states (except for the final state that has no outgoing arcs). For large \( a \), and depending on \( s \), the worst-case complexity is:

\[
O(f(x)) = \begin{cases} 
O \left( \left( \frac{a}{s-1} \right)^{s-1} \right) & \text{for } a > e(s - 1) \\
O \left( \sqrt[4]{e} \right) = O \left( e^{\frac{e}{a}} \right) < O \left( e^{s-1} \right) & \text{for } a \leq e(s - 1)
\end{cases}
\]

Acknowledgments

We would like to thank Jean-Marc Champarnaud for his advice.

References

Reducing the Time Complexity of Testing for Local Threshold Testability

Avraham Trakhtman
Bar-Ilan University, Dep. of Math. and CS
52900 Ramat Gan, Israel
trakht@macs.biu.ac.il

Abstract. A locally threshold testable language $L$ is a language with the property that for some nonnegative integers $k$ and $l$, whether or not a word $u$ is in the language $L$ depends on (1) the prefix and suffix of the word $u$ of length $k - 1$ and (2) the set of intermediate substrings of length $k$ of the word $u$ where the sets of substrings occurring at least $j$ times are the same, for $j \leq l$. For given $k$ and $l$ the language is called $l$-threshold $k$-testable. A finite deterministic automaton is called threshold locally testable if the automaton accepts a $l$-threshold $k$-testable language for some $l$ and $k$.

New necessary and sufficient conditions for a deterministic finite automaton to be locally threshold testable are found. On the basis of these conditions, we modify the algorithm to verify local threshold testability of the automaton and reduce the time complexity of the algorithm. The algorithm is implemented as a part of C/C++ package TEstAS (testability of automata and semigroups).

Keywords: Automaton, threshold locally testable, graph, algorithm

Introduction

The locally threshold testable languages were introduced by Beauquier and Pin [1] and have now different applications [8], [13], [14], [15]. In particular, stochastic locally threshold testable languages, also known as $n$-grams are used in pattern recognition and in speech recognition, both in acoustic-phonetics decoding as in language modelling [13]. These languages generalize the concept of local testability [2], [7]. The last can be considered as a special case of local $l$-threshold testability for $l = 1$.

Necessary and sufficient conditions of local testability and a polynomial-time algorithms for local testability problem based on these conditions were described [5], [6], [11], [10] and implemented [3], [4], [11], [12]. Necessary and sufficient conditions of local threshold testability for DFA and a polynomial-time algorithm of order $O(n^5)$ for local threshold testability problem were found in [11]. The algorithm was implemented [11], [12]. We modify these necessary and sufficient conditions for the transition graph of a deterministic finite automaton and reduce in that way the order of the algorithm for local threshold testability.
to \(O(n^4)\). By \(n\) is denoted here the sum of the nodes and edges of the transition graph of the automaton (\(n\) can be also considered as the product of the number of states by the size of alphabet).

The considered algorithm is now implemented as a part of \(C/C++\) package TESTAS replacing the old version of the algorithm \([11]\) and reducing the time of the execution. An important verification tool of the package is a possibility of parallel checking of both syntactic semigroup and transition graph of the automaton. This possibility was intensively used in the case of the considered algorithm. The package TESTAS (testability of automata and semigroups) contains a set of procedures for deciding whether or not a language given by its minimal automaton or by syntactic semigroup of the minimal automaton is locally testable, threshold locally testable, left locally testable, right locally testable, piecewise testable and some other programs \([12]\).

**Notation and Definitions**

Let \(\Sigma\) be an alphabet and let \(\Sigma^+\) denote the free semigroup on \(\Sigma\). If \(w \in \Sigma^+\), let \(|w|\) denote the length of \(w\). Let \(k\) be a positive integer. Let \(i_k(w)\) \((t_k(w))\) denote the prefix [suffix] of \(w\) of length \(k\) or \(w\) if \(|w| < k\). Let \(F_{k,j}(w)\) denote the set of factors of \(w\) of length \(k\) with at least \(j\) occurrences. A language \(L\) is called \(k\)-**threshold** \(k\)-**testable** if there is an alphabet \(\Sigma\) such that for all \(u, v \in \Sigma^+\), if \(i_k-1(u) = i_k-1(v), t_k-1(u) = t_k-1(v)\) and \(F_{k,j}(u) = F_{k,j}(v)\) for all \(j \leq l\), then either both \(u\) and \(v\) are in \(L\) or neither is in \(L\).

An automaton is \(l\)-**threshold** \(k\)-**testable** if the automaton accepts a \(l\)-threshold \(k\)-testable language.

A language \(L\) [an automaton] is **locally threshold testable** if it is \(l\)-threshold \(k\)-testable for some \(k\) and \(l\).

The number of nodes of the graph \(\Gamma\) is denoted by \(|\Gamma|\).

The direct product of \(i\) copies of the graph \(\Gamma\) is denoted by \(\Gamma^i\). The edge \(p_1, \ldots, p_n \to q_1, \ldots, q_n\) in \(\Gamma^i\) is labelled by \(\sigma\) iff for each \(i\) the edge \(p_i \to q_i\) in \(\Gamma\) is labelled by \(\sigma\).

A maximal strongly connected component of the graph will be denoted for brevity as \(\text{SCC}\), a finite deterministic automaton will be denoted as \(\text{DFA}\).

A node from oriented graph will be called \(\text{C - node}\) if the node belongs to some oriented cycle. \(\text{C - node}\) can be defined also as a node that has right unit in the transition semigroup of the automaton.

An oriented labelled graph is complete if every node of the graph has for every label from alphabet of labels an outgoing edge with this label. A non-complete graph can be completed by adding sink node and then adding lacking edges from corresponding nodes to sink node.

If an edge \(p \to q\) is labelled by \(\sigma\) then let us denote the node \(q\) as \(p\sigma\).

We shall write \(p \preceq q\) if the node \(q\) is reachable from the node \(p\) or \(p = q\).

In the case \(p \succeq q\) and \(q \succeq p\) we write \(p \sim q\) \((p\) and \(q\) belong to one \(\text{SCC}\) or \(p = q\)).
1 The Necessary and Sufficient Conditions of Local Threshold Testability

Let us formulate the result of Beauquier and Pin [1] in the following form:

**Theorem 1.1** [1] A language $L$ is locally threshold testable if and only if the syntactic semigroup $S$ of $L$ is aperiodic and for any two idempotents $e, f$ and elements $a, u, b$ of $S$ we have

$$eafuebf = ebueaf$$

(1)

**Lemma 1.2** [11] Let the node $(p, q)$ be an $C$-node of $\Gamma^2$ of a locally threshold testable DFA with state transition graph $\Gamma$ and suppose that $p \sim q$.

Then $p = q$.

**Lemma 1.3** Let the nodes $(q, t_1), (q, t_2)$ be $C$-nodes of the graph $\Gamma^2$ of a locally threshold testable DFA with state transition graph $\Gamma$ and transition semigroup $S$.

Suppose that $(q, t_1) \succeq (q, t_2)$ and $q \succeq t_1$.

Then $t_1 \sim t_2$.

**Proof.** We have $(q, t_1)e = (q, t_1), (q, t_2)e_2 = (q, t_2), (q, t_1)a = (q, t_2), qu = t_1$ for some idempotents $e, e_2$ and elements $a, u \in S$. By theorem 1.1, $eueae_2e = eae_2e$.

Now $t_2e = t_2e_2e = t_1eae_2e = qeueae_2e = qeae_2e = t_1e = t_1$.

So $t_2 \succeq t_1$. We have $t_1a = t_2$, whence $t_1 \sim t_2$.

**Lemma 1.4** Let $p, q, r_1$ be nodes of graph $\Gamma$ of a locally threshold testable DFA such that $(p, r_1)$ is an $C$-node, $p \succeq q$ and for some node $r, t$ the node $(r, t)$ is an $C$-node, $p \succeq r \succeq r_1$ and $(p, r_1) \succeq (q, t)$.

Then $q \succeq t$.

**Proof.** We have $(p, r_1)e = (p, r_1)$ for some idempotent $e$ from the transition semigroup $S$ of the automaton, $(r, t)i = (r, t)$ for some idempotent $i \in S$, $(p, r_1)b = (q, t), pa = r$ and $ru = r_1$ for some elements $a, u, b \in S$.

In view of identity (1) of local threshold testability $t = peaiuebi = pebiueai = qiuebi$, whence $q \succeq t$.

**Définition 1.5** Let $p, q, r_1$ be nodes of graph $\Gamma$ such that $(p, r_1)$ is an $C$-node, $p \succeq q$ and for some node $r, (q, r)$ is an $C$-node and $p \succeq r \succeq r_1$.

For such nodes $p, q, r_1$ let $T3_{SCC}(p, q, r_1)$ be the SCC of $\Gamma$ containing the set

$$T(p, q, r_1) := \{ t \mid (p, r_1) \succeq (q, t), q \succeq t \text{ and } (q, t) \text{ is an } C\text{-node}\}$$

$T3_{SCC}$ is not well defined in general case, but an another situation holds for local threshold testability.
Lemma 1.6 Let the nodes \((q, r), (p, r_1), (q, t_1), (q, t_2)\) be C-nodes of the graph \(T^2\) of a locally threshold testable DFA with state transition graph \(T\) and transition semigroup \(S\), \((p, r_1) \supseteq (q, t_1), q \supseteq t_i\) for \(i = 1, 2\) and \(p \supseteq r \supseteq r_1\).

Then \(t_1 \sim t_2\).

\(T_{3\text{SCC}}(p, q, r_1)\) of a locally threshold testable DFA is well defined (nodes from \(T_{3\text{SCC}}(p, q, r_1)\) belong to one SCC).

Proof. We have \((p, r_1) e = (p, r_1)\) for some idempotent \(e\) from the transition semigroup \(S\), \((q, r)f = (q, r), (q, t_1)f_1 = (q, t_1), (q, t_2)f_2 = (q, t_2)\) for some idempotents \(f, f_1, f_2 \in S\), \((p, r_1)b_1 = (q, t_1), (p, r_1)b_2 = (q, t_2), p a = r\) and \(r u = r_1\) for some elements \(a, u, b_1, b_2 \in S\).

For given nodes \(p, q, r_1\), let us form set \(R\) of nodes \(r\) such that \((q, r)\) is an \(C\)-node, \(p a = r\) and \(ru_1 = r_1\) for some elements \(a_1, u_1 \in S\) and the set \(T\) of nodes \(t\) such that \((p, r_1) \supseteq (q, t)\) and \((q, t)\) is an \(C\)-node. So \(t_1, t_2 \in T\) and \(r \in R\).

For any \(r \in R\) and any \(t \in T\) let us consider the node \(t f\) where idempotent \(f\) is right unit for \((q, r)\). The nodes \((q, t)\) and \((q, t f)\) are \(C\)-nodes, \(q \supseteq t, \(q, t f) \supseteq \(q, t)\), whence by lemma 1.3, \(t f \sim t\) for any such \(f\). Therefore the set \(T\) can be divided on subsets from different not comparable SCC such that in any subset for any \(r\) exists a node \(t_r\) such that the node \((q, r, t_r)\) is an \(C\)-node.

Let us consider for given \(r\) two such nodes \(t_1\) and \(t_2\) from different SCC. Let \(f\) be common right unit for triple \(q, r, t_1\). The identity of local threshold testability (1) implies \(t_1 = r_1 b_1 f = r u e b_1 f = p e a f u e b_1 f = p e b_1 f u e a f\). We have \(p b_1 = p b_2 = q\), whence \(t_1 = p e b_1 f u e a f = p e b_2 f u e a f = p e a f u e b_2 f = t_2 f\). So \(t_2 \supseteq t_1\). Analogously, \(t_1 \supseteq t_2\), whence \(t_2 \sim t_1\). Thus \(T\) is not divided between distinct SCC.

So \(T_{3\text{SCC}}(p, q, r_1)\) of a locally threshold testable DFA is well defined.
Lemma 1.7 Suppose that \((p, r_1)\) and \((p, r_2)\) are \(C\)-nodes of the graph \(\Gamma^2\) of a locally threshold testable DFA with state transition graph \(\Gamma\) and transition semigroup \(S\), \(q \succeq t_i\), \(p \succeq s \succeq r_i\) (\(i = 1, 2\)) for some \(s\) such that \((q, s)\) is an \(C\)-node.

Then from \(r_1 \sim r_2\) follows \(t_1 \sim t_2\) and \(T_3SCC(p, q, r_1) = T_3SCC(p, q, r_2)\).

**Proof.** We have \((p, r_1)e_1 = (p, r_1)\), \((p, r_2)e_2 = (p, r_2)\), \((q, s)f = (q, s)\), for some idempotents \(e_1, e_2, f \in S\), \(p \succeq s\) and \((p, r_1)b_1 = (q, t_1)\), \((p, r_2)b_2 = (q, t_2)\), \(su_i = r_i\) for some elements \(u_i, b_1, b_2 \in S\).

From \((p, r_1)e_1 = (p, r_1)e_2\), by lemma 1.3, follows \(r_1 \sim r_1e_2\). Notice that \(r_2e_2 = r_2 \sim r_1\), whence \(r_2 \sim r_1e_2\). Therefore, by lemma 1.2, \(r_2 = r_1e_2\). Then \((p, r_1)e_2b_2 = (q, r_1e_2b_2) = (q, r_2b_2) = (q, t_2)\), whence in view of \((p, r_1)b_1 = (q, t_1)\), we have \((p, r_1) \succeq (q, t_1)\) and \((p, r_1) \succeq (q, t_2)\). Now by lemma 1.6, we have \(t_1 \sim t_2\). So the nodes \(t_1, t_2\) belong to \(T_3SCC(p, q, r_1)\). Let us notice that the node \(t_2\) belongs to \(T_3SCC(p, q, r_2)\) too. Therefore \(T_3SCC(p, q, r_1) = T_3SCC(p, q, r_2)\).

Lemma 1.8 Suppose that in a locally threshold testable DFA with state transition graph \(\Gamma\) and transition semigroup \(S\) holds

- the nodes \((p, q_1)\) and \((q, r)\) are \(C\)-nodes of the graph \(\Gamma^2\),
- \(p \succeq q\) and \(p \succeq r\),
- there exists a node \(r_1\) such that \((q, r) \succeq (q_1, r_1)\) and \((p, r_1)\) is an \(C\)-node.

Then \(T_3SCC(p, q, r_1)e = T_3SCC(p, q, r_1)\) for any idempotent \(e \in S\) such that \(pe = p\).

**Proof.** We have \(p \succeq r \succeq r_1\). Then \((p, r_1) \succeq (p, r_1)e = (p, r_1)e\) and both these nodes are \(C\)-nodes. Therefore, by lemma 1.3, \(r_1e \sim r_1\). Lemma 1.7 implies now \(T_3SCC(p, q, r_1)e = T_3SCC(p, q, r_1)\).

**Theorem 1.9** DFA \(A\) with state transition complete graph \(\Gamma\) (or completed by sink state) is locally threshold testable iff

1) for every \(C\)-node \((p, q)\) of \(\Gamma^2\) \(p \sim q\) implies \(p = q\),
2) for every four nodes \(p, q, t, r_1\) of \(\Gamma\) such that
- the node \((p, r_1)\) is an \(C\)-node,
- \((p, r_1) \succeq (q, t)\),
- there exists a node \(r\) such that \(p \succeq r \succeq r_1\) and \((r, t)\) is an \(C\)-node.
holds \( q \geq t \).

3) every \( T_{3SCC} \) is well defined,

4) for every four nodes \( p, q, r, q_1 \) of \( \Gamma \) such that

- the nodes \((p, q_1)\) and \((q, r)\) are \( C \)-nodes of the graph \( \Gamma^2 \),
- \( p \geq q \) and \( p \geq r \),
- there exists a node \( r_1 \) such that \((q, r) \geq (q_1, r_1)\) and \((p, r_1)\) is an \( C \)-node

holds \( T_{3SCC}(p, q, r_1) = T_{3SCC}(p, r, q_1) \).

**Proof.** Let \( A \) be locally threshold testable. Condition 1 follows in this case from lemma 1.2. Condition 2 follows from lemma 1.4. Condition 3 follows from lemma 1.6 and from the definition 1.5 of \( T_{3SCC} \).

Let us check the last condition. For some idempotent \( e \) holds \( (p, q_1)e = (p, q_1) \). By lemma 1.8, \( T_{3SCC}(p, q_1) = T_{3SCC}(p, q_1) \). Therefore we compare below \( T_{3SCC}(p, q_1) \) and \( T_{3SCC}(p, r, q_1) \).

We have \( t_1 = r_1 e b f = e b f e a f. \) Then by (1) \( e b f e a f = e b f e a f \). So \( t_1 \geq t \). Analogously, \( t \geq t_1 \). Therefore \( t_1 = t \), whence \( T_{3SCC}(p, r, q_1) = T_{3SCC}(p, q_1) \). Consequently \( T_{3SCC}(p, r, q_1) = T_{3SCC}(p, r, q_1) \).

Suppose now that all four conditions of the theorem hold. Our aim is to prove the local threshold testability of DFA. For this aim, let us consider an arbitrary node \( v \), arbitrary elements \( a, u, b \) and idempotents \( e, f \) from syntactic semigroup \( S. \) We are to prove that \( v e a f = e b f e a f \) (theorem 1.1).

Let us denote \( p = v e, q = v e b f, q_1 = v e b f e, t = v e b f e a f, r_1 = v e a f, t_1 = v e a f \).

We have \( (p, r_1) \geq (q, t), (p, r_1) \) and \((r_1, t)\) are \( C \)-nodes, \( p \geq r \geq r_1 \), therefore by condition 2 \( q \geq t \). Now according to the definition of \( T_{3SCC} \) we have \( t \in T_{3SCC}(p, q_1) \) and \( t_1 \in T_{3SCC}(p, r, q_1) \). The node \((p, r_1)\) is an \( C \)-node and \((q, r)u = (q_1, r_1)\). Therefore the condition 4 implies \( T_{3SCC}(p, q_1) = T_{3SCC}(p, r, q_1) \). These sets are well defined, whence by condition 3, \( t_1 = t \).

Both these nodes have common right unit \( f \). Consequently, \( (t, t_1) \) is an \( C \)-node. Now by condition 1, \( t_1 = t \). Thus \( v e a f = e b f e a f \) is true for arbitrary node \( v \) and the identity \( e a f = e b f e a f \) of local threshold testability holds.

It remains now to prove the aperiodicity of \( S \). Let \( p \) be an arbitrary node from \( \Gamma \) and let \( s \) be an arbitrary element of \( S \). \( S \) is finite, whence for some
Integers $k$ and $m$ hold $s^k = s^{k+m}$. Let us consider the nodes $ps^k$ and $ps^{k+1}$. So $ps^k \geq ps^{k+1}$ and in view $s^k = s^{k+m} = s^{k+1} s^{m-1}$ we have $ps^{k+1} \geq ps^k$. Thus $ps^{k+1} \sim ps^k$. Some power of $s$ is an idempotent and a unit for both these nodes. Then by condition 1, $ps^k = ps^{k+1}$. Since the syntactic semigroup $S$ is aperiodic.

So the automaton is locally threshold testable.

**Lemma 1.10** Let the graph $\Gamma$ be the transition graph of a locally threshold testable DFA with transition semigroup $S$. Let $P(q, r)$ be non-empty set of $C$-nodes $p$ such that $p \geq q$ and $p \geq r$ for $C$-node $(q, r)$.

Suppose $r_2 \rho r_1$ if $(q, r) \geq (q_1, r_1)$ and $(q, r) \geq (q_2, r_2)$ for some $C$-nodes $(p, r_1), (p, r_2)$ and let $\Sigma$ be transitive closure of the relation $\rho$.

Then $r_1 \Sigma r_2$ implies $T_3SCC(p, q, r_1) = T_3SCC(p, q, r_2)$ for any $p \in P(q, r)$.

**Proof.** We have $(q, r)f = (q, r), (q, r)u_1 = (q_1, r_1), (q, r)u_2 = (q_1, r_2), p(a) = q, p(b) = r$ for some idempotents $e, f$ and elements $u_i, a, b$ from $S$.

So $q_1 = peafu_2 = peafu_1, pebfu_1 = r_1, pebfu_2 = r_2$. For $r_1eaf$ from $T_3SCC(p, q, r_1)$ holds $r_1eaf = pebfu_1 eaf = peafu_1ebf = peafu_1ebf = pebfu_2 eaf = r_2eaf \in T_3SCC(p, q, r_2)$. So by definition $T_3SCC(p, q, r_1) = T_3SCC(p, q, r_2)$. Thus $r_2 \rho r_1$ implies $T_3SCC(p, q, r_1) = T_3SCC(p, q, r_2)$. By lemma 1.8, $T_3SCC(p, q, r_1) = T_3SCC(p, q, r_1)$ and therefore $T_3SCC(p, q, r_1) = T_3SCC(p, q, r_2)$.

The nodes $q, r$ are used in this lemma in a completely symmetrical manner, whence we can conclude that from $(q, r) \geq (q_1, r_1)$ and $(q, r) \geq (q_2, r_2)$ follows $T_3SCC(p, r, q_1) = T_3SCC(p, r, q_2)$. By theorem 1.9, they both are equal to $T_3SCC(p, q, r_1)$. By lemma 1.8, $T_3SCC(p, q, r_1) = T_3SCC(p, q, r_1)$. Therefore the transition closure $\Sigma$ of the relation $\rho$ keeps $T_3SCC$ too.

## 2 An Algorithm to Check the Local Threshold Testability

A linear depth-first search algorithm finding all $SCC$ (see [9]) will be used. By $n$ will be denoted the sum of the nodes and edges of the graph, $g > 0$ denotes the size of alphabet. Let us notice that $|\Gamma| |g + 1| \geq n$. The algorithm is based on the theorem 1.9 for complete graph $\Gamma$ (or $\Gamma$ completed by sink state).

Let us find all $SCC$ of the graphs $\Gamma$ and $\Gamma^2$ and all their $C$-nodes. Let us recognize the reachability on the graph $\Gamma$ and form the table of reachability for all pairs of $\Gamma$. The step uses $O(|\Gamma|^2 g)$ time and space.

Let us check the first condition of the theorem 1.9 (see lemma 1.2 too). For every $C$-node $(p, q)$ $(p \neq q)$ from $\Gamma^2$ let us check the condition $p \sim q$. A negative answer for any considered node $(p, q)$ implies the validity of the condition. In opposite case the automaton is not locally threshold testable. The time of the step is $O(|\Gamma|^2)$.

Let us go to the second condition of the theorem 1.9. For every $C$-node $(p, r_1)$ we form the set $T$ of nodes $t \in \Gamma$ such that $r_1 \geq t$ and for some node $r$ holds: the node $(r, t)$ is an $C$-node and $p \geq r \geq r_1$. If there exists a node $q$ such that $(p, r_1) \geq (q, t)$ for $t \in T$ and $q \neq t$, the automaton is not threshold
locally testable. It is a step of worst case asymptotic cost $O(|\Gamma|^4g)$ with space complexity $O(|\Gamma|^3)$.

Now let us check the condition 3 of the theorem 1.9 for well defined $T3_{SCC}$. For every three nodes $p, q, r_1$ of the graph $\Gamma$ such that $(p, r_1)$ is a C-node, $p \succeq r_1$ and $p \succeq q$, let us find a node $r$ such that $p \succeq r \succeq r_1$ and then let us find $T3_{SCC}$ (see definition 1.5). In case that $T3_{SCC}$ is not well defined (for $t_1, t_2 \in T3_{SCC} t_1 \not\sim t_2$), by lemma 1.6, the automaton is not threshold locally testable. The time required for this step in the worst case is $O(|\Gamma|^4g)$. The space complexity is $O(|\Gamma|^3)$.

Let us use the assertion of the lemma 1.10. For every C-node $(q, r)$ of the graph $\Gamma^2$ let us form the set $P(q, r)$ of C-nodes $p$ such that $p \succeq q$ and $p \succeq r$. For non-empty set $P(q, r)$ we continue. For every node $q_1$ let us form the set $Rq_1$ of nodes $r_1$ such that $(q, r) \succeq (q_1, r_1)$ and the last is a C-node. Let us unite the sets $Rq_1$ with non-empty intersection. So we obtain a family of disjoint sets defined by relation $\Upsilon$ from lemma 1.10. If for two nodes $r_1, r_2$ from one set and for some $p$ from $P(q, r)$ such that $(p, r_1)$ and $(p, r_2)$ are C-nodes holds $T3_{SCC}(p, q, r_1) \neq T3_{SCC}(p, q, r_2)$ the automaton is not threshold locally testable (lemma 1.10). If $T3_{SCC}(p, q, r_1) \neq T3_{SCC}(p, r, q_1)$ where $(p, q_1)$ is an C-node, then the automaton is also not threshold locally testable according to the third condition of the theorem 1.9. The time required for this step in the worst case is $O(|\Gamma|^4g)$ with $O(|\Gamma|^3)$ space.

A positive answer for all considered cases implies in view of theorem 1.9 the local threshold testability of the automaton.

The time complexity of the algorithm to check local threshold testability is no more than $O(|\Gamma|^4g)$. The space complexity of the algorithm is $\max(O(|\Gamma|^2g), O(|\Gamma|^3))$. In more ordinary formulation we have $O(n^4)$ time and $O(n^3)$ space. The program worked essentially faster in many cases we have studied because of structure of the algorithm.

References


Branching Automata with Costs –
A Way of Reflecting Parallelism in Costs

Dietrich Kuske and Ingmar Meinecke

Institut für Algebra, Technische Universität Dresden
D-01062 Dresden, Germany
{kuske,meinecke}@math.tu-dresden.de

Abstract. Extending work by Lodaya and Weil, we propose a model of branching automata with costs in which the calculation of the cost of a parallel composition is handled differently from the calculation of the cost of a sequential composition. Our main result characterizes the behavior of these automata in the spirit of Kleene’s and Schützenberger’s theorems.

The technical report [12] that this extended abstract is based on contains complete proofs and can be accessed at the net.

1 Introduction

This paper reports on our research into parallel systems with costs in the setting of series-parallel posets. One of its roots is the line of research initiated by Grabowski [7] and Gischer [6]. They extended previous ideas by Kleene on sequential systems build by nondeterministic choice, iteration and sequential composition. Gischer proposed, in order to model parallel systems, in addition a parallel composition. It turned out that series-parallel posets are ideally suited to describe executions of such systems. Later, Lodaya and Weil [14, 13] proposed a finite-state device capable of accepting series-parallel posets. These automata model parallelism by branching - hence the name “branching automata”.

Suppose we wanted to calculate the minimal duration of a run in a modularly constructed system. The execution time of a sequential composition is the sum of the durations, and that of a parallel composition is the maximum of the durations of the arguments, possibly increased by some duration for the necessary fork and join operations at the beginning and end. A given series-parallel poset can be executed in different ways and we should consider the minimal duration of all possible executions. In order to accompany this situation, we introduce bisemirings, i.e., structures consisting of two semirings on a joint domain with the same additive operation. Costs of executions in our model of branching automata with costs are then evaluated in such bisemirings and the behavior of a branching automaton with costs is a function that associates with any series-parallel poset an element from the bisemiring.

It is the aim of this paper to characterize those functions that are associated with branching automata with costs. For this, we employ and extend the machinery from the theory of weighted automata (see [16, 10, 2, 9] for expositions).

© Springer-Verlag Berlin Heidelberg 2003
In this field, one starts from a nondeterministic finite (word or tree) automaton and provides its transitions with weights, costs or multiplicities (depending on the community). M. Droste raised the question whether branching automata can be provided with costs in a semiring. Our conceptional contribution in this respect is the observation that one should not just consider cost structures with one multiplication, but that several multiplications are necessary to model the phenomena of parallelism.

We characterize the behavior of branching automata with costs in the style of theorems by Kleene [8] and Schützenberger [17] stating the equivalence of regularity and rationality. Several related results are known: for trees, there is a wealth of results of different generality [1, 9, 5, 15]; for Mazurkiewicz traces, Droste and Gastin proved a theorem à la Schützenberger [3]; and for infinite words, Droste and Kuske showed a result in the spirit of the theorems by Büchi and by Schützenberger [4]. When Lodaya and Weil considered languages accepted by branching automata, they observed that unbounded parallelism cannot be captured completely by rational operations, their main results hold for languages of bounded width [13]. Since, in a parallel system, the width corresponds to the number of independent processes, this boundedness restriction seems natural to us. Therefore and similarly, our characterization holds for branching automata with costs only that generate functions with support of bounded width. For this class, we get as our main result the equivalence of acceptance by branching automata with costs and rationality (see Theorem 5.2).

2 Basic Definitions

Let \( \Sigma \) be a finite alphabet. A \( \Sigma \)-labeled poset \((V, \leq, \tau)\) is a finite poset\(^1\) \((V, \leq)\) equipped with a labeling function \( \tau : V \rightarrow \Sigma \). The width \( \text{wd}(t) \) of \( t \) is the maximal size of a subset of \( V \) whose elements are mutually incomparable.

The sequential product \( t_1 \cdot t_2 \) of \( t_1 = (V_1, \leq_1, \lambda_1) \) and \( t_2 \) is the \( \Sigma \)-labeled poset

\[
(V_1 \cup V_2, \leq_1 \cup (V_1 \times V_2) \cup \leq_2, \tau_1 \cup \tau_2).
\]

Graphically, \( t_2 \) is put on top of \( t_1 \). The parallel product \( t_1 \parallel t_2 \) is defined as \((V_1 \cup V_2, \leq_1 \cup \leq_2, \tau_1 \cup \tau_2)\), i.e., the two partial orders are put side by side. SP denotes the least class of \( \Sigma \)-labeled posets containing all labeled singletons and closed under the application of the sequential and the parallel product, its elements are series-parallel posets or sp-posets for short. We say that \( t \) is sequential if it cannot be written as a parallel product \( t = u \parallel v \). Dually, \( t \) is called parallel if it cannot be written as a sequential product \( t = u \cdot v \) of \( u, v \in \text{SP} \). The only sp-posets which are both sequential and parallel are the singleton posets that we identify with the elements of \( \Sigma \). By Gischer [6], every \( t \in \text{SP} \) admits a maximal parallel factorization \( t = t_1 \parallel \ldots \parallel t_n \) (which is unique up to commutativity) where \( n \geq 1 \) and each \( t_i \in \text{SP} \) \((i = 1, \ldots, n)\) is sequential, and a unique maximal sequential decomposition \( t = t'_1 \cdot \ldots \cdot t'_m \) where \( m \geq 1 \) and each \( t'_i \in \text{SP} \) \((i = 1, \ldots, m)\).

\(^1\) In this paper, we consider even only nonempty posets.
is parallel. Hence, SP is freely generated by $\Sigma$ subject to associativity of both operations and commutativity of the parallel product.

**Definition 2.1.** A bisemiring $\mathbb{K} = (K, \oplus, \circ, 0, 1)$ is a set $K$ equipped with three binary operations called addition $\oplus$, sequential multiplication $\circ$ and parallel multiplication $\circ$ such that:

1. $(K, \oplus, 0)$ is a commutative monoid, $(K, \circ, 1)$ a monoid, and $(K, \circ)$ a commutative semigroup,
2. both $\circ$ and $\circ$ distribute over $\oplus$, and
3. $0$ is absorbing for $\circ$ and $\circ$, i.e., $k \circ 0 = 0 \circ k = k \circ 0 = 0$ for all $k \in K$.

The structure $(K, \oplus, \circ, 0, 1)$ is a semiring. Moreover, $(K, \oplus, \circ, 0)$ is almost a semiring; only the parallel multiplication does not have to admit a unit. Any commutative semiring can be seen as a bisemiring by identifying sequential and parallel multiplication. Under these trivial bisemirings we only mention the Boolean bisemiring $\mathbb{B} = (\{0, 1\}, \lor, \land, \land, 0, 1)$.

**Example 2.2.** The structure $(\mathbb{R} \cup \{+\infty\}, \min, +, \max, +, 0)$ is a bisemiring that we referred to in the introduction. Here, $0$ is the unit for the sequential multiplication $+$ and $+\infty$ is the absorbing zero of the bisemiring.

Let $a \in \Sigma$. We interpret $a$ as some action and assume $a$ has a duration of $\text{time}(a)$. Let $\text{time}(a) = +\infty$ if $a$ cannot be performed. For any $t = t_1 \cdot \ldots \cdot t_n \in \text{SP}$ we put $\text{time}(t) = \text{time}(t_1) + \ldots + \text{time}(t_n)$, and for $t = t_1 || \ldots || t_m \in \text{SP}$ we put $\text{time}(t) = \max\{\text{time}(t_1), \ldots, \text{time}(t_m)\}$. Hence, $\text{time} : (\text{SP}, \cdot, ||) \rightarrow (\mathbb{R} \cup \{+\infty\}, +, \max)$ is a homomorphism and can be interpreted as the duration time of an sp-poset $t$. In Example 3.2, we will present an automaton that computes the minimal execution time of an sp-poset using the semiring $(\mathbb{R} \cup \{+\infty\}, \min, +, \max, +, 0)$.

**Example 2.3.** Let $\Sigma$ be a fixed finite alphabet and let $\text{SP}^1 = \text{SP} \cup \{\varepsilon\}$ where $\varepsilon$ acts as unit w.r.t. $\cdot$ and $||$. Then the class of sp-languages $(\mathcal{P}(\text{SP}^1), \cup, \cdot, ||, \emptyset, \{\varepsilon\})$ is a bisemiring. Here the multiplications $\cdot$ and $||$ are defined elementwise.

## 3 Branching Automata with Costs

In this section we introduce a model of automata generalizing the concept of branching automata by Lodaya and Weil [13] and their behavior. We fix an alphabet $\Sigma$ and a bisemiring $\mathbb{K}$. By $\mathcal{P}_2(Q)$ we denote the collection of subsets of $Q$ of cardinality 2.

**Definition 3.1.** A branching automaton with costs from $\mathbb{K}$ over the alphabet $\Sigma$, or a BRAC for short, is a tuple $A = (Q, T_{\text{seq}}, T_{\text{fork}}, T_{\text{join}}, \lambda, \gamma)$ where

- $Q$ is a finite set of states,
- $T_{\text{seq}} : Q \times \Sigma \times Q \rightarrow K$ is the sequential transition function,
- $T_{\text{fork}} : Q \times \mathcal{P}_2(Q) \rightarrow K$ is the fork transition function,
- $T_{\text{join}} : \mathcal{P}_2(Q) \times Q \rightarrow K$ is the join transition function,
- $\lambda : Q \rightarrow \Sigma$ the input function,
- $\gamma : Q \rightarrow \mathbb{K}$ the output function.
$\lambda, \gamma : Q \to K$ are the initial and the final cost function, respectively.

We write $p \xrightarrow{a} q$ if $T_{\text{seq}}(p, a, q) = k \neq 0$ and call it a sequential transition; if it only matters that the costs are distinct from 0, we write $p \xrightarrow{a} q$. Similarly, we write $p \xrightarrow{k} \{p_1, p_2\}$ and $p \xrightarrow{} \{p_1, p_2\}$ if $T_{\text{fork}}(p, \{p_1, p_2\}) = k \neq 0$. In the same way, we understand $\{q_1, q_2\} \xrightarrow{k} q$ and $\{q_1, q_2\} \xrightarrow{} q$. A state $q \in Q$ is an initial state if $\lambda(q) \neq 0$. Dually, $q$ is a final state if $\gamma(q) \neq 0$.

3.1 The Behavior of Branching Automata

In order to calculate the cost of an sp-poset $t \in \text{SP}$ in a BRAC $\mathcal{A}$ we introduce the notion of a path in $\mathcal{A}$. We consider labeled directed graphs $G = (V, E, \nu, \eta)$ with a unique source $\text{src}(G)$ and a unique sink $\text{sk}(G)$ where $\nu : V \to Q$ is a total function and $\eta : E \to \Sigma$ is a partial function. The labeled graph $G$ with $V = \{v_1, v_2\}$, $E = \{(v_1, v_2), v_1 \xrightarrow{1, 2} p_i \text{ and } v_2 \xrightarrow{} q\}$ and the cost $\text{cost}(G) = \nu(q)$ is the union of both graphs where the sink of $G_1$ that is a partial function. The labeled graph $G$ is given by summing up $\nu$.

The labeled graph $G$ is a path with vertices $V = V_1 \cup V_2 \cup \{u, w\}$ and edges $E = E_1 \cup E_2 \cup \{(u, \text{src}(G_1)), \text{sk}(G_1), w \mid i = 1, 2\}$. For $v \in V$ we put $\nu(v) = \nu_i(v)$ and $\nu(w) = q$, and $\eta = \eta_1 \cup \eta_2$. The sequential product is associative, and every $p,q$-parallel product is commutative, but not associative.

The set PT($\mathcal{A}$) of paths of the BRAC $\mathcal{A}$ is the smallest set of labeled directed graphs $G$ that contains all atomic paths of $\mathcal{A}$ and is closed under the sequential product and under all $p,q$-parallel products with $p, q \in Q$ as defined above.

Inductively, we define the label $\text{lab}(G) \in \text{SP}$ and the cost $\text{cost}(G) \in K$ for any path $G$. For an atomic path $G : p \xrightarrow{a} q$, we put $\text{lab}(G) = a$ and $\text{cost}(G) = T_{\text{seq}}(p, a, q)$. Further, $\text{lab}(G_1 \parallel G_2) = \text{lab}(G_1) \cdot \text{lab}(G_2)$, $\text{cost}(G_1 \parallel G_2) = \text{cost}(G_1) \circ \text{cost}(G_2)$, and $\text{cost}(G_1 \parallel p,q G_2) = \text{cost}(G_1) \| \text{lab}(G_2)$. To define the cost of $G = G_1 \parallel p,q G_2$, let $p_i = \nu_i(\text{src}(G_i))$ and $q_i = \nu_i(\text{sk}(G_i))$ for $i = 1, 2$. Then

$$\text{cost}(G) = T_{\text{fork}}(p, \{p_1, p_2\}) \circ \left[ \text{cost}(G_1) \circ \text{cost}(G_2) \right] \circ T_{\text{join}}(\{q_1, q_2\}, q).$$

The cost of such a parallel path can be interpreted as follows. At first we have a cost for branching the process, then the cost for the two subprocesses and finally the cost for joining the subprocesses. These costs come up one after the other and, therefore, are multiplied sequentially. On the other hand, the costs of the two subprocesses are multiplied in parallel.

We denote by $G : p \xrightarrow{t} q$ that $G$ is a path from state $p$ to state $q$ with label $t \in \text{SP}$. Then the cost of some $t \in \text{SP}$ from $p$ to $q$ in $\mathcal{A}$ is given by summing up
the costs of all possible paths from \( p \) to \( q \) with label \( t \):

\[
\text{cost}_{p,q}(t) = \bigoplus_{G:p \rightarrow q} \text{cost}(G) .
\]

The cost of \( t \in \text{SP} \) in \( \mathcal{A} \) is defined as

\[
(S(\mathcal{A}), t) = \text{cost}_{\mathcal{A}}(t) = \bigoplus_{p,q \in \text{Q}} \lambda(p) \circ \text{cost}_{p,q}(t) \circ \gamma(q) .
\]

Then \( S(\mathcal{A}) : \text{SP} \rightarrow \mathbb{K} \) is the behavior of \( \mathcal{A} \) or, equivalently, is recognized by \( \mathcal{A} \). A function \( S : \text{SP} \rightarrow \mathbb{K} \) is regular if there is a BRAC \( \mathcal{A} \) such that \( S = S(\mathcal{A}) \).

**Example 3.2.** In this example, we define a branching automaton with costs \( \mathcal{A} \) whose behavior measures the height of a pomset, i.e., \( (S(\mathcal{A}), t) = \text{height}(t) \) for any sp-pomset \( t \). For this to work, we use the bisemiring \( (\mathbb{R} \cup \{+\infty\}, \min, +, \max) \) from Example 2.2. The automaton has just three states \( p_0, p_1, p_2 \). Any of these states can fork into the other two at cost 0; similarly, any two distinct of these states can be joined into the remaining one at cost 0. In any state, we can execute any action at cost 1 (without changing the state). Figure 1 depicts a run of this BRAC on the sp-poset \( t = (aa \parallel b)(a \parallel bb) \). In that picture, join- and fork-transitions are indicated by a semi-circle between the edges involved. Next to these semi-circles, we denote the cost of the corresponding transition. The path is the sequential product of two “bubbles” whose costs we calculate first. The first bubble is the parallel product of an atomic \( b \)-transition and the sequential \( aa \)-path. Since the join- and fork-transitions involved in this product have cost 0, the cost of a bubble is \( 0 + \max(1 + 1, 1) + 0 = 2 \). Since this holds for both bubbles, the total cost is \( 2 + 2 = 4 \) which equals the height of the poset \( (aa \parallel b)(a \parallel bb) \).

If any action is executed in one unit of time, then the height of the poset measures the execution time of the sp-poset. This example can be refined in such a way that atomic actions can have different execution times that can even depend on the state they are executed in. Using the bisemiring \( (\mathbb{R} \cup \{-\infty, +\infty\}, \min, \max, +) \), the same BRAC as above computes the width of a poset (i.e., the maximal number of parallel sub-processes). In [12], we give another example of a BRAC, this time over the bisemiring of subsets of \( \Sigma^* \). It calculates, from an sp-poset \( t \), the set of words that label a maximal linearly ordered subset of \( t \).
3.2 Formal Power Series over SP-Posets

To characterize the possible behavior of branching automata with costs, we introduce the notion of formal power series over sp-posets with values in a bisemiring. This concept is both a generalization of the well-known formal power series over words (cf. [16]) and the sp-languages as introduced by Lodaya and Weil [13].

A formal power series over SP with values in the bisemiring \( \mathbb{K} \) or sp-series is a function \( S : SP \rightarrow \mathbb{K} \). With \( (S, t) := S(t) \), it is written as a formal sum:

\[
S = \sum_{t \in SP} (S, t)t.
\]

The support of \( S \) is \( \text{supp} S := \{ t \in SP \mid (S, t) \neq 0 \} \). Formal power series with support a singleton are called monomials. The class of all formal power series over \( SP \) with values in \( \mathbb{K} \) is denoted by \( \mathbb{K} \langle\langle SP \rangle\rangle \).

Now we introduce some operations for sp-series. Let \( S, T \in \mathbb{K} \langle\langle SP \rangle\rangle \). We define:

1. the sum \( S + T \) by \( (S + T, t) := (S, t) \oplus (T, t) \),
2. the scalar products \( k \cdot S \) and \( S \cdot k \) for \( k \in \mathbb{K} \) by \( (k \cdot S, t) := k \circ (S, t) \) and \( (S \cdot k, t) := (S, t) \odot k \),
3. the sequential product \( S \cdot T \) by \( (S \cdot T, t) := \bigoplus_{t = u \cdot v} (S, u) \odot (T, v) \) where the sum is taken over all sequential factorizations \( t = u \cdot v \) with \( u, v \in SP \),
4. the parallel product \( S \parallel T \) by \( (S \parallel T, t) := \bigoplus_{(u, v) : t = u \parallel v} (S, u) \odot (T, v) \) where we add over all pairs \( (u, v) \) such that \( t = u \parallel v \) with \( u, v \in SP \). Because of the commutativity of \( \parallel \) in SP, both \( (S, u) \odot (T, v) \) and \( (S, v) \odot (T, u) \) contribute to the sum.
5. the sequential iteration \( S^+ \) of an sp-series \( S \) by

\[
(S^+, t) := \bigoplus_{1 \leq n \leq |t|} \bigoplus_{t = u_1 \cdot \ldots \cdot u_n} (S, u_1) \odot \ldots \odot (S, u_n)
\]

where we sum up over all possible sequential factorizations of \( t \).

Collectively, we refer to these operations as the series-rational operations of \( \mathbb{K} \langle\langle SP \rangle\rangle \).

Similarly to the sequential iteration, one can define the parallel iteration. Already in the theory of sp-languages, this parallel iteration causes severe problems [14]. Smoother results are obtained if one does not allow the parallel iteration in rational expressions [13, 11].

The operations \( +, \cdot, \parallel \) are associative on \( \mathbb{K} \langle\langle SP \rangle\rangle \), and \( + \) and \( \parallel \) are even commutative. The series \( 0 \) with \( (0, t) = 0 \) for all \( t \in SP \) is the unit w.r.t. \( + \) and absorbing w.r.t. \( \cdot \) and \( \parallel \).

The class \( \mathbb{K}^{\text{ser-rat}} \langle\langle SP \rangle\rangle \) of series-rational sp-series over \( \Sigma \) with values in \( \mathbb{K} \) is the smallest class containing all monomials that is closed under the series-rational operations of \( \mathbb{K} \langle\langle SP \rangle\rangle \).
Let $\mathbb{K}$ and $\mathbb{K}'$ be bisemirings, $h : \mathbb{K} \to \mathbb{K}'$ a bisemiring homomorphism, and $f : \mathbb{S}P \to \mathbb{S}P$ a function that commutes with $\cdot$ and $\parallel$. Further, let $S$ be an sp-series over $\mathbb{K}$. For $t \in \mathbb{S}P$, define $(\overline{h}(S), t) = h(S, t)$ and $(\overline{f}(S), t) = \bigoplus_{f(s)=t} (S, s)$.

**Proposition 3.3.** The functions $\overline{h}$ and $\overline{f}$ commute with the series-rational operations. In particular, they preserve series-rationality.

We consider as a special case the Boolean bisemiring $\mathbb{B}$. An sp-language $L$ is a subset of $\mathbb{S}P$. Any sp-language $L \subseteq \mathbb{S}P$ can be identified with its **characteristic series** $1_L$ where $\text{supp} 1_L = L$. This isomorphism maps the class $\mathbb{B}^{\text{rat}}\langle \langle \mathbb{S}P \rangle \rangle$ to the class of series-rational sp-languages $\mathbb{S}P^{\text{rat}}$ (cf. [13] for the definition). Therefore, the theory of sp-series is a generalization of the theory of sp-languages as investigated by Lodaya and Weil [13].

An sp-language $L \subseteq \mathbb{S}P$ has **bounded width** if there exists an integer $n$ such that for each element $t \in L$ we have $\text{wd}(t) \leq n$. Similarly, we call $S \in \mathbb{K}[\langle \langle \mathbb{S}P \rangle \rangle]$ **width-bounded** if $\text{supp} S$ has bounded width. From the definition of series-rational sp-series we get immediately that any series-rational sp-series has bounded width. As for sp-languages the opposite is not true.

### 3.3 Bounded Width and Bounded Depth

The bounded width of a regular sp-series is reflected by the “bounded depth” of a BRAC. Every atomic path is of depth 0, $\text{dp}(G_1 \cdot G_2) = \max\{\text{dp}(G_1), \text{dp}(G_2)\}$ and $\text{dp}(G_1 \parallel_{p,q} G_2) = 1 + \max\{\text{dp}(G_1), \text{dp}(G_2)\}$. Therefore, the depth of a path measures the nesting of branchings within the path. A BRAC $\mathcal{A}$ is of **bounded depth** if the depth of its paths is uniformly bounded (Lodaya and Weil [13] require this for successful paths, only). Any series recognized by a BRAC of bounded depth is of bounded width. The converse for sp-languages was shown in [11] by just counting and thereby limiting the depth of a path. That proof can be extended to bisemirings that do not allow an additive decomposition of 0. The problem in the general case arises from the existence of paths of different depths having the same label $t$. Then two such paths can have non-zero costs, but the sum of these costs can be 0. If now the path with larger depth is disabled, the cost of the sp-poset $t$ changes (see the complete paper [12] for a more elaborated example). To overcome this problem, we will keep track of the actual width (and not just the depth) of a poset. This is achieved by a stack where the widths encountered up to the last fork transition are stored. In addition, we restrict the size of the stack as well as the natural numbers stored therein to the width of the support. This allows to perform the construction within the realm of finite-state systems. By $S|_n$ we denote the restriction of $S$ to the sp-posets of width less than or equal to $n$.

**Proposition 3.4.** Let $n \in \mathbb{N}$ and $S$ a regular sp-series. Then $S|_n$ can be recognized by a depth-bounded BRAC.
Proof. Let $\mathcal{A}$ be a BRAC recognizing $S$. We put $[n] := \{1, \ldots, n\}$, and $[n]^+$ denotes the set of nonempty words over $[n]$. From $\mathcal{A}$ we construct a new automaton $\mathcal{A}'$ as follows. The states get a new component that we prefer to think of as a stack because of the operations that will be performed on it; its alphabet is $[n]$ and its size is restricted to depth $n$:

$$Q' = \{(p, w) \in Q \times [n]^+ : |w| \leq n\}.$$ 

A sequential transition does not change the stack:

$$T'_\text{seq}((p, u), (a, q, v)) = \begin{cases} T_{\text{seq}}(p, a, q) & \text{if } u = v \\ 0 & \text{otherwise} \end{cases},$$

but at any fork, we push a new top symbol onto the stack (if this is not forbidden by the size restriction):

$$T'_\text{fork}((p, u), \{ (p_1, u_1), (p_2, u_2) \}) = \begin{cases} T_{\text{fork}}(p, \{p_1, p_2\}) & \text{if } u_1 = u_2 = u1 \\ 0 & \text{otherwise} \end{cases}.$$

The real work is done in a join transition. First, the two top symbols are summed up – they are meant to measure the width of the two subpaths, hence their sum is the width of the total path since the matching fork. Then, this sum is compared with the previous symbol on the stack (i.e., the width of the path between the previous fork and join) and the larger of the two replaces them both:

$$T'_\text{join}(\{ (q_1, v_1), (q_2, v_2) \}, (q, v)) = T_{\text{join}}(\{q_1, q_2\}, q) \text{ if there are } w \in [n]^+ \text{ and } x, y_1, y_2, z \in [n] \text{ such that } v_1 = wxy_1, v_2 = wxy_2, v = wz \text{ with } z = \max\{x, y_1 + y_2\}. \text{ Otherwise, } T'_\text{join}(\{ (q_1, v_1), (q_2, v_2) \}, (q, v)) = 0.$$

At the start, the newly constructed BRAC places a 1 in its stack and it is only allowed to stop if the stack contains precisely one symbol:

$$\lambda'(p, u) = \begin{cases} \lambda(p) & \text{if } u = 1 \\ 0 & \text{otherwise} \end{cases}, \quad \gamma'(q, v) = \begin{cases} \gamma(q) & \text{if } |v| = 1 \\ 0 & \text{otherwise} \end{cases}.$$

The construction ensures that the symbol in the stack of a final state is the width of the poset executed by the automaton. \hfill \Box

Let $S$ be a regular sp-series of bounded width. Then, as a consequence of this proposition, it can be recognized by a BRAC of bounded depth. Thus, width-boundedness of regular sp-series and depth-boundedness of BRACs are equivalent notions.

4 Closure Properties of Regular SP-Series

In the proofs of the following results, we use branching automata with restricted possibilities to enter and to leave the automaton. A BRAC $\mathcal{A}$ is called normalized if there are unique states $i$ and $f$ with $\lambda(i) \neq 0$, $\gamma(f) \neq 0$ and, moreover, we have for all $p, p_1, p_2 \in Q$ and $a \in \Sigma$
\(- \lambda(i) = 1 \) and \( T_{\text{seq}}(p, a, i) = 0 \) as well as \( T_{\text{join}}(\{p_1, p_2\}, i) = 0 = T_{\text{fork}}(p_1, \{i, p_2\}) \),

\(- \gamma(f) = 1 \) and \( T_{\text{seq}}(f, a, p) = 0 \) and \( T_{\text{fork}}(f, \{p_1, p_2\}) = 0 = T_{\text{join}}(\{p_1, f\}, p_2) \).

**Proposition 4.1.** Any regular sp-series is the behavior of some normalized BRAC.

The closure of regular sp-series under \( + \) is shown by the classical construction for the union of regular languages: one considers the disjoint union of two automata. Closure under parallel composition is shown similarly: one considers the disjoint union of two BRACs and adds initial fork- and terminal join-transitions as appropriate. It turns out that this obvious idea works for normalized BRACs, only, which is, by the proposition above, no restriction on the sp-series.

**Proposition 4.2.** Let \( S_1 \) and \( S_2 \) be regular sp-series. Then \( S_1 + S_2 \) and \( S_1 \parallel S_2 \) are regular.

The rest of this section is devoted to the closure of the class of regular series under sequential multiplication and sequential iteration. It is tempting to believe that constructions familiar from the theory of finite automata work here as well. But, as already observed for sp-languages by Lodaya and Weil [13], the obvious variant of the classical construction for the product does not yield the correct result. The problem is that the newly constructed automaton can switch from \( A_1 \) into \( A_2 \) independently in parallel subpaths.

Lodaya and Weil showed that this problem does not arise when one restricts to “behaved automata”. Then they show that one can transform any branching automaton into an equivalent behaved one. We proceed differently giving a direct construction for the sequential product. More precisely, we “send a signal” from the initial state along the path. In fork transitions, this signal is only propagated along one branch. In order not to duplicate paths, the signal is sent to the “smaller” of the two states that arise from the fork transition.\(^2\) The newly constructed BRAC can only switch from \( A_1 \) into \( A_2 \) in the presence of this signal, and in any successful path, the signal has to be present at the final state.

**Proposition 4.3.** Let \( S_1, S_2 \in \mathbb{K}\langle\langle \text{SP} \rangle\rangle \) be two regular sp-series. Then \( S_1 \cdot S_2 \) is regular.

Similarly to the sequential composition, the classical construction for the sequential iteration suggests itself - and yields an incorrect result as the following example shows.

**Example 4.4.** We work with the Boolean bisemiring \( \mathbb{B} = (\{0, 1\}, \lor, \land, \land, 0, 1) \), i.e., in the setting of sp-languages. Consider the BRAC from Figure 2 (left) where we omitted the costs; any transition depicted has cost 1 and no further transitions.

\(^2\) This is actually the reason why we have to assume that these two states are different, i.e., that we work with sets in the definition of fork- and join-transitions and not with multisets as Lodaya and Weil do.
have nonzero cost. The support of the recognized sp-series is \( \{a \parallel b, dae\} \). The classical construction for the sequential iteration tells us to add, among other transitions, one of the form \( q_1 \xrightarrow{e} i \) since there is a sequential transition \( q_1 \xrightarrow{e} f \) in the BRAC in consideration. But then we get the path depicted in Figure 2 (right) whose label is \( (acda)\parallel b \) which does not belong to the sequential iteration of the sp-language generated by the BRAC we started with.

Lodaya and Weil’s solution is, again, to use behaved automata. Our direct construction sends not just one, but two signals. These two signals travel along different ways: whenever they can separate in a fork transition, they do so. Then the newly constructed automaton is allowed to jump from the final state to the initial state only in case both signals are present. As before, in any successful path, both signals are present in the first and the last state.

**Proposition 4.5.** If \( S \in \mathbb{K}\langle\langle SP \rangle\rangle \) is regular, then the sequential iteration \( S^+ \) is regular.

**Proof.** Let \( A \) be a branching automaton with costs with behavior \( S \). As explained above, we need two signals that travel along maximal ways in a path. This is modeled by extending the states of \( A \) by two binary digits, i.e., we set \( Q' = Q \times \{0, 1\}^2 \). In sequential transitions, the signals are simply propagated:

\[
T'_{\text{seq}}((p, x, x'), a, (q, y, y')) = \begin{cases} T'_{\text{seq}}(p, a, q); & \text{if } y = x, y' = x' \text{ otherwise} \\ 0 & \text{otherwise} \end{cases}
\]

In a fork transition, the first signal is propagated to the smaller successor state and the second signal to the larger one. In particular, they separate in case both are present. For that, let \( \leq \) be an arbitrary but fixed linear order on \( Q \), and assume \( p_1 \leq p_2 \). Then we put:

\[
T'_{\text{fork}}((p, x, x'), \{(p_1, x_1', x_1'), (p_2, x_2', x_2')\})
\]

\[
= \begin{cases} T_{\text{fork}}(p, \{p_1, p_2\}); & \text{if } p_1 \neq p_2, x_1 = x, \\ x_1' = 0, x_2 = 0, x_2' = x' \text{ otherwise} \\ 0 & \text{otherwise} \end{cases}
\]
Similarly to the sequential transitions, in a join transition both incoming signals are propagated to the unique successor state:

$$T'_{\text{join}}(\{(q_1, x_1, x'_1), (q_2, x_2, x'_2)\}, (q, x, x')) = \begin{cases} T_{\text{join}}(\{q_1, q_2\}, q) & \text{if } x'_1 = 0, x_2 = 0, x = x_1, x' = x'_2 \\ 0 & \text{otherwise} \end{cases}.$$  

Finally, the automaton starts in the presence of both signals and is only allowed to finish when both signals are present:

$$\lambda'(p, x, x') = \begin{cases} \lambda(p) & \text{if } x = x' = 1 \\ 0 & \text{otherwise} \end{cases}, \quad \gamma'(q, x, x') = \begin{cases} \gamma(q) & \text{if } x = x' = 1 \\ 0 & \text{otherwise} \end{cases}.$$  

Due to Proposition 4.1, $A$ can be assumed to be normalized. Then $A'$ is normalized too. For any sequential or join transition of $A'$ with final state $f'$, we add the corresponding transition into $i'$. The construction above ensures that any path of this new BRAC $A^+$ that contributes to the cost is the sequential product of finitely many paths of the BRAC $A$. Moreover, this gives a bijection between the sets of successful paths of $A^+$ and tuples of successful paths of $A$.

\begin{theorem}
Let $K$ be an arbitrary bisemiring. Every series-rational sp-series $S \in K\langle\langle SP\rangle\rangle$ is regular; it is even recognized by a normalized BRAC of bounded depth.
\end{theorem}

5 From Regular and Bounded Depth to Series-Rational

If $G = G_1 \|_{p,q} G_2$ is a path, $f$ denotes the starting fork transition of $G$, and $j$ the finishing join transition of $G$, then we say that $(f, j)$ is a matched pair.

\begin{theorem}
The behavior of any BRAC of bounded depth is series-rational.
\end{theorem}

\begin{proof}
Let $\sqsubseteq$ be a relation on the set of matched pairs such that $(f, j) \sqsubseteq (f', j')$ whenever there exists a parallel path $G$ starting with $f'$ and ending with $j'$ that contains $(f, j)$ as a matched pair. Since $A$ is of bounded depth, $\sqsubseteq$ can be extended to a linear order $\leq$.

Let $(f, j)$ be a matched pair and $p, q \in Q$ states of $A$. We denote by $S^{(f, j)}_{p, q}$ the series with

$$(S^{(f, j)}_{p, q}, t) = \bigoplus_{G: p \stackrel{t}{\rightarrow} q} \text{cost}(G)$$

where $t \in SP$ and the paths $G$, over which the sum extends, use only matched pairs smaller than or equal to $(f, j)$. By induction along $\leq$, one shows that $S^{(f, j)}_{p, q}$ is series-rational which yields the result since there are only finitely many matched pairs.
\end{proof}
The special case $\mathbb{K} = \mathbb{B}$ was shown by Lodaya and Weil [13]. Their proof uses a nested induction which we simplified here to just one induction along the linear order of matched pairs.

Now we can prove the main theorem about regular and series-rational sp-series.

**Theorem 5.2.** Let $\mathbb{K}$ be an arbitrary bisemiring and $S \in \mathbb{K}\langle\langle SP\rangle\rangle$. The following are equivalent:

1. $S$ is series-rational.
2. $S$ is recognized by a BRAC of bounded depth.
3. $S$ is regular and has bounded width.

**Proof.** Due to Theorem 4.6, (1) implies (2). By Theorem 5.1, (2) implies (1). Statement (3) implies (2) by Proposition 3.4, the remaining implication “(2) $\Rightarrow$ (3)” is obvious. $\Box$

By putting $\mathbb{K} = \mathbb{B}$, we get as a consequence of the last theorem the result of Lodaya and Weil [13] for sp-languages.

**References**


New Complexity Results for Some Linear Counting Problems Using Minimal Solutions to Linear Diophantine Equations
(Extended Abstract)

Gaoyan Xie, Cheng Li, and Zhe Dang

School of Electrical Engineering and Computer Science, Washington State University
Pullman, WA 99164, USA
zdang@eecs.wsu.edu

Abstract. The linear reachability problem is to decide whether there is an execution path in a given finite state transition system such that the counts of labels on the path satisfy a given linear constraint. Using results on minimal solutions (in nonnegative integers) for linear Diophantine systems, we obtain new complexity results for the problem, as well as for other linear counting problems of finite state transition systems and timed automata. In contrast to previously known results, the complexity bounds obtained in this paper are polynomial in the size of the transition system in consideration, when the linear constraint is fixed.

1 Introduction

Model-checking [7, 22] is a technique that automatically verifies a finite state transition system against a temporal property usually specified in, e.g., Computation Tree Logic (CTL) [7] or Linear Temporal Logic (LTL) [20], by exhaustively exploring the finite state space of the system. The usefulness of model-checking has been demonstrated by several successful model-checkers (e.g., SMV [17], SPIN [15], BMC [3]) which have been used to test/verify industrial-level hardware/software systems with significant sizes.

Although both CTL and LTL are expressive, many temporal properties are out of their scope. For instance, event counting is a fundamental concept to specify some important fairness properties. As a motivating example, we consider the design (depicted as a finite state transition system \( A \) in Figure 1) of a process scheduler. The scheduler schedules two kinds of processes: \( P_r \) and \( P_w \) according to some scheduling strategy. A transition with label \( P_r \) (resp. \( P_w \)) is taken when the scheduler chooses a \( P_r \) (resp. \( P_w \)) process to run. It is required that the design shall satisfy some fairness properties; e.g., starting from state \( s_0 \), whenever \( s_0 \) is reached, the number of \( P_r \) processes scheduled is greater than or equal to the number of \( P_w \) processes scheduled and less than or equal to twice the number of \( P_w \) processes scheduled. To ensure that the design meets the requirement, we need check whether for any path \( p \) that starts from and ends with \( s_0 \), the
Fig. 1. An example of a scheduler

linear constraint, \(#P_w(p) \leq #P_r(p) \leq 2#P_w(p)\), is satisfied, where \(#P_w(p)\) (resp. \(#P_r(p)\)) stands for the count of labels \(P_w\) (resp. \(P_r\)) on path \(p\). Notice that this property is nonregular \([6]\) and, since the counts could go unbounded, the property is not expressible in CTL or LTL.

In general, by considering its negation, the property can be formulated as a linear reachability problem as follows.

- **Given:** A finite state transition system \(A\) with labels \(a_1, \ldots, a_k\), two designated states \(s_{\text{init}}\) and \(s_{\text{final}}\), and a linear constraint \(U(x_1, \ldots, x_k)\).
- **Question:** Is there a path \(p\) of \(A\) from \(s_{\text{init}}\) to \(s_{\text{final}}\) such that \(p\) satisfies \(U\) (i.e., \(U(#a_1(p), \ldots, #a_k(p))\) holds)?

The reachability problem is decidable. To see this, one can treat \(A\) as a finite automaton with initial state \(s_{\text{init}}\) and final state \(s_{\text{final}}\). \(L(A)\) is the regular language over alphabet \(\{a_1, \ldots, a_k\}\) accepted by \(A\). A naive decision procedure consists of the following three steps: (i) Compute a regular expression for \(L(A)\), (ii) Calculate the semilinear set of the regular expression defined by a Presburger formula \(R\) \([19]\), and (iii) Check the satisfiability of the Presburger formula \(R \land U\).

Unfortunately, the time complexity of this procedure is at least \(O(2^{|S|})\), where \(|S|\) is the number of states in \(A\), even when \(k\) is fixed. This is because the size of the regular expression, in worst cases, is exponential in \(|S|\) \([16]\). In this paper, we present a new algorithm solving the linear reachability problem. Our algorithm is completely different from the naive one. In the algorithm, we estimate a bound \(B\) (called a bounding box) from \(A\) and \(U\) such that, the **Question**-part is true iff the truth is witnessed by some \(p\) on which the count \(#a_i(p)\) for each label \(a_i\) is bounded by \(B\). Interestingly, after a complex loop analysis, estimating a bounding box \(B\) is reduced to a number theory problem: finding bounds for nonnegative minimal solutions to linear Diophantine systems. There has been much research on the latter problem for homogeneous/inhomogeneous systems with (nonnegative) integer solutions \([4, 5, 13, 21]\). Suppose that \(U\) is in a disjunctive normal form over linear equations/inequalities. Using the Borosh-Flahive-Treybig bound in \([4]\), we are able to show that, in worst cases and when \(|S|\) is \(\gg\) the size (which will be made clear later) of \(U\), the bounding box \(B\) is bounded by \(O(|S|^{k+L+3})\), where \(L\) is the maximal number of conjunctions in
a single disjunctive term of $U$. Using this bounding box, one can easily show that the linear reachability problem is solvable in time

$$O(|S|^{2k(k+L+3)+2}),$$

(1)

when $|S| \gg k$ and the size of $U$. In particular, when $k$ and $U$ are fixed, the complexity is polynomial in $|S|$. This is in contrast to the complexity of the naive algorithms that is exponential in the state number $|S|$.

Our complexity result in (1) for the linear reachability problem can be further used to obtain complexity bounds (which were unknown) for some other linear counting problems involving linear constraints over counts. For instance, we consider a linear liveness problem as follows. For an $\omega$-path $\pi$ of $A$, we say that $\pi$ is $U$-i.o. (infinitely often) at state $s'$ if there are infinitely many prefixes $p$ of $\pi$ such that $p$ ends at $s'$ and satisfies $U$. The liveness problem is to decide, given two states $s$ and $s'$, whether $A$ has an $\omega$-path $\pi$ that starts from $s$ and is $U$-i.o. at $s'$. The application issues of this problem can be found in [11]. In particular, in the same paper, it is shown that the liveness problem is decidable. However, the time complexity was unknown. In this paper, we are able to use (1) to establish a complexity bound for the liveness problem, which is similar to the bound given in (1).

We also consider the linear reachability problem when $A$ is ordered; i.e., on any path $p$ from $s_{\text{init}}$ to $s_{\text{final}}$, each label $a_j$ is after all the $a_i$'s, whenever $i < j$. For this restricted model of $A$, we obtain a smaller (than (1)) complexity bound $O(|S|^{4k-1})$ for the reachability problem, by using the Pottier bound in [21] (the Borosh-Flahive-Treybig bound is not applicable here). Interestingly, this restricted model and the complexity bound can be used to establish a new complexity result for timed automata [1]. We first consider discrete timed automata where clocks take integral values. The linear reachability problem for discrete timed automata is defined as follows:

- **Given:** A discrete timed automaton $D$, two designated states $s_{\text{init}}$ and $s_{\text{final}}$, and two linear constraints $U$ and $U'$ over $k$ variables.

- **Question:** are there clock values $v_1, \ldots, v_k, v_1', \ldots, v_k'$ such that $\langle s_{\text{init}}, v_1, \ldots, v_k \rangle$ reaches $\langle s_{\text{final}}, v_1', \ldots, v_k' \rangle$ and both $U(v_1, \ldots, v_k)$ and $U'(v_1', \ldots, v_k')$ hold?

Though many temporal verification problems involving linear constraints over clocks are known to be undecidable [1, 2, 12], the linear reachability problem is decidable for timed automata (even when the clocks are dense) [8, 9, 10]. However, an upper bound for the worst case complexity was unknown. Using the result for the linear reachability problem when $A$ is ordered, we can establish that the linear reachability problem for discrete timed automata is solvable in time $O(|S|^{8k})$ when $|S| \gg k$, the sizes of $U$ and $U'$, and the maximal absolute value of all the constants appearing in the clock constraints of $A$. This result can be generalized to timed automata with dense clocks using the pattern technique in [9]. In this extended abstract, all the proofs are omitted. For a complete exposition, see the full version of this paper at http://www.eecs.wsu.edu/~zdang.
2 Preliminaries

Let $N$ be the set of nonnegative integers and $k$ be a positive integer. A finite state transition system can be defined as

$$A = (S, \Sigma, E),$$

where $S$ is a finite set of states, $\Sigma = \{a_1, \cdots, a_k\}$ is a set of labels, $E \subseteq S \times (\Sigma \cup \{\epsilon\}) \times S$ is a set of transitions. When $E \subseteq S \times \{\epsilon\} \times S$, the system is called a finite state machine. A path $p$ of $A$ is a finite sequence of transitions in the form of

$$(s_0 \tau_1 s_1) \cdots (s_i \tau_i s_{i+1}) \cdots (s_n-1 \tau_{n-1} s_n)$$

for some $n$ such that for each $0 \leq i < n$, $(s_i \tau_i s_{i+1}) \in E$. Path $p$ is a simple cycle if $s_0, \ldots, s_{n-1}$ are distinct and $s_0 = s_n$. Path $p$ is a simple path if $s_0, \ldots, s_{n-1}, s_n$ are all distinct. For any path $p$ of $A$, let $\#(p)$ denote the $k$-ary vector $(\#_{a_1}(p), \cdots, \#_{a_k}(p))$, where each $\#_{a_i}(p)$ stands for the number of label $a_i$'s occurrences on $p$, $1 \leq i \leq k$.

Let $x_1, \cdots, x_k$ be nonnegative integer variables. An atomic linear constraint is in the form of

$$b_1 x_1 + \cdots + b_k x_k \sim b$$

where $\sim \in \{=, \geq\}$, $b_1, \ldots, b_k$ and $b$ are integers. When $\sim = =$ (resp. $\geq$), the constraint is called an equation (resp. inequality). The constraint is made homogeneous if one makes $b = 0$ in the constraint. A linear constraint $U$ is a Boolean combination of atomic linear constraints (using $\land, \lor, \neg$). Without loss of generality, throughout this paper, we assume that the linear constraint $U$ is always written as a disjunction $U_1 \lor \cdots \lor U_m$, for some $m$, of conjunctions of atomic linear constraints. When $m = 1$, $U$ is called a conjunctive linear constraint. $U$ is made homogeneous if each atomic linear constraint in $U$ is made homogeneous; we use $U^{\text{hom}}$ to denote the result. In particular, a conjunctive linear constraint $U$ is a linear Diophantine equation system if each atomic linear constraint in $U$ is an equation.

Suppose that $U$ is a conjunctive linear constraint, which contains $e$ equations and $l - e$ inequalities. One may write $U$ into $Bx \sim b$, where $\sim \in \{=, \geq\}$, $B$ ($l$ by $k$) and $b$ ($l$ by $1$) are matrices of integers, and $x$ is the column of variables $x_1, \cdots, x_k$. As usual, $(B, b)$ is called the augmented matrix of $U$, and $B$ is called the coefficient matrix of $U$. We use $\|B\|_{1, \infty}$ to denote $\max_i \{|\sum_j |B_{ij}|\}$ ($b_{ij}$ is the element of row $i$ and column $j$ in $B$) and use $\|b\|_\infty$ to denote the maximum of the absolute values of all the elements in $b$. Assume $r$ is the rank of $(B, b)$, and $\Gamma_1$ (resp. $\Gamma_2$) is the maximum of the absolute values of all the $r \times r$ minors of $B$ (resp. $(B, b)$).

When $U$ is a linear Diophantine equation system (i.e., $e = l$), for any given tuples $(v_1, \ldots, v_k)$ and $(v'_1, \ldots, v'_k)$ in $\mathbb{N}^k$, we say $(v_1, \ldots, v_k) \leq (v'_1, \ldots, v'_k)$ if $v_i \leq v'_i$ for all $1 \leq i \leq k$. We say $(v_1, \ldots, v_k) < (v'_1, \ldots, v'_k)$ if $(v_1, \ldots, v_k) \leq (v'_1, \ldots, v'_k)$ and $v_i < v'_i$ for some $1 \leq i \leq k$. A tuple $(v'_1, \ldots, v'_k)$ is a minimal solution to $U$ if $(v'_1, \ldots, v'_k)$ is a solution to $U$ but any $(v_1, \ldots, v_k)$ with
(0, \cdots, 0) < (v_1, \cdots, v_k) < (v'_1, \cdots, v'_k)$ is not. Clearly, there are only finitely many minimal solutions to $U$. It has been an active research area to estimate a bound for minimal solutions, and the following Borosh-Flahive-Treybig bound [4] is needed in this paper.

**Theorem 1.** (Borosh-Flahive-Treybig bound) A linear Diophantine equation system $U$ has solutions in nonnegative integers iff it has a solution $(x_1, \cdots, x_k)$ in nonnegative integers, such that $r$ unknowns are bounded by $\Gamma_1$ and $k - r$ unknowns are bounded by $(\max(k, l) - r + 1)\Gamma_2$.

The Borosh-Flahive-Treybig bound gives a bound for one of the minimal solutions in nonnegative integers to the inhomogeneous system $U$. In contrast, the following Pottier bound gives an upper bound for all of the “minimal solutions” to a conjunctive $U$ (which is not necessarily a linear equation system); this result can be simply obtained from Corollary 1 in [21].

**Theorem 2.** (Pottier bound) For any conjunctive linear constraint $U$ that contains $e$ equations and $l - e$ inequalities, there are two finite sets $S$ and $S^{\text{hom}} = \{v_1, \cdots, v_q\}$, for some $q$, of vectors in $\mathbb{N}^k$ such that

- each element in $S$ (resp. $S^{\text{hom}}$) is a solution to $U$ (resp. $U^{\text{hom}}$),
- for any $\mathbf{v} \in \mathbb{N}^k$, $\mathbf{v}$ is a solution to $U$ iff there are $t_1, \cdots, t_q \in \mathbb{N}$, $\mathbf{v} = \mathbf{v}_0 + t_1\mathbf{v}_1 + \cdots + t_q\mathbf{v}_q$ for some $\mathbf{v}_0 \in S$,
- each component of all the vectors in $S \cup S^{\text{hom}}$ is bounded by $(2 + \|B\|_{1, \infty} + \|b\|_{\infty})^{k+l+e}$.

Therefore, for a conjunctive linear constraint $U$, every solution can be represented as the sum of a small solution and a nonnegative linear combination of small solutions to $U^{\text{hom}}$ (clearly, the inverse is also true). Here, “small” means that the solutions are bounded by the Pottier bound.

When $U$ is a linear constraint (i.e., $m \geq 1$), the Pottier bound of $U$ is defined to be the maximal of all the bounds obtained from Theorem 2 for each conjunctive linear constraint in $U$.

An inequality can be translated into an equation by introducing a slack variable (e.g., $x_1 - 2x_2 \geq 3$ into $x_1 - 2x_2 - u = 3$ where $u$, a new variable on $\mathbb{N}$, is the slack variable). So if $U$ is a conjunctive linear constraint (in which there are $e$ equations and $l - e$ inequalities) over $x_1, \cdots, x_k$, we may write $U$ into an equation system $U(x_1, \cdots, x_k, x_{k+1}, x_{k+l-1})$ with $l$ equations, where $x_{k+1}, x_{k+l-1}$ are the slack variables.

In the next section, we will derive a bounding box $B$ for the linear reachability problem such that its Question-part is true iff the truth is witnessed by a $p$ satisfying $\#_a, (p) \leq B$ for each $1 \leq i \leq k$. From this $B$, the time complexity for solving the linear reachability problem can be easily obtained.
3 A Bounding Box $B$
for the Linear Reachability Problem

Let $\mathcal{A}$ be a finite state transition system specified in (2). A set of $k$-ary nonnegative integer vectors $Q$ is a small linear set (wrt the given $\mathcal{A}$) if $Q$ is in the form of

$$\{e_0 + \sum_{1 \leq j \leq r} X_j e_j : \text{each } X_j \geq 0\},$$  \hspace{1cm} (5)

where nonnegative integer $r$ satisfies

$$r \leq |S|^k,$$  \hspace{1cm} (6)

$k$-ary nonnegative integer vectors $e_0, ..., e_r$ satisfy

$$e_0 \leq |S|^2 \cdot 1,$$  \hspace{1cm} (7)

and for each $j = 1, ..., r,$

$$e_j \leq |S| \cdot 1,$$  \hspace{1cm} (8)

where 1 stands for the identity vector. $Q$ is a small semilinear set if it is a union of finitely many small linear sets.

Recall that the linear reachability problem is to decide whether there exists a path $p$ in $\mathcal{A}$ from $s_{\text{init}}$ to $s_{\text{final}}$ such that $p$ satisfies a given linear constraint $U(x_1, ..., x_k)$. Let $\mathcal{P}$ be all paths of $\mathcal{A}$ from $s_{\text{init}}$ to $s_{\text{final}}$. We use $\#(\mathcal{P})$ to denote the set of $k$-ary nonnegative integer vectors $\{\#(p) : p \in \mathcal{P}\}$. Using a complex loop analysis technique by reorganizing simple loops on a path, one can show that $\#(\mathcal{P})$ is a small semilinear set.

**Lemma 1.** $\#(\mathcal{P})$ is a small semilinear set. That is, it can be represented as, for some $t$,

$$\#(\mathcal{P}) = \bigcup_{1 \leq i \leq t} Q_i$$  \hspace{1cm} (9)

where each $Q_i$ is a small linear set in the form of (5).

**Remark 1.** One might have already noticed that, (9) and (5) appear nothing new, since they simply rephrase the known fact that $\#(\mathcal{P})$ defines a semilinear set [19]. However, the bounds for the coefficients shown in (6, 7, 8) are new.

Now let’s turn to the property formula $U$. Recall that $U$ is written as a disjunction of $m$ conjunctive linear constraints

$$U = \bigvee_{1 \leq i \leq m} U_i.$$  \hspace{1cm} (10)

\footnote{Note that though $t$ may be large, it is irrelevant here.}
Fix any $1 \leq i \leq m$. Suppose that $U_i$ contains $l$ atomic linear constraints. After adding (at most $l$) slack variables $y_1, \ldots, y_l$, $U_i$ can be written into the following form:

$$
\begin{aligned}
  b_{11} x_1 + \ldots + b_{1k} x_k + g_1 y_1 &= b_1 \\
  &\vdots \\
  b_{l1} x_1 + \ldots + b_{lk} x_k + g_l y_l &= b_l
\end{aligned}
$$

where the $b$'s and $g$'s are integers (each $g$ is -1 or 0). Let $B$ be the coefficient matrix for variables $x_1, \ldots, x_k$ and $b$ be the column of $b_1, \ldots, b_l$ in (11). Define $w_1 = ||B||_{1,\infty}$ and $w_2 = ||b||_{\infty}$. We may assume $w_1 > 0$ (otherwise let $w_1 = 1$). In the sequel, we shall use the following notions: $W_1$ (the maximum of all the values $w_1$ among all $U_i$'s), $W_2$ (the maximum of all the values $w_2$ among all $U_i$'s), and $L$ (the maximum of all the values $l$ among all $U_i$'s).

Due to the disjunctive representations of (10) and (9), we can consider only one conjunction of $U$ in the form of (11) and only one linear set in the form of (5). That is, by substituting $x = (x_1, \ldots, x_k)$ in (11) with the expression in (5): $x = e_0 + \sum_{1 \leq j \leq r} X_j e_j$, the equation system (11) is transformed into the following equation system with unknowns $X_1, \ldots, X_r$ and $y_1, \ldots, y_l$:

$$
\begin{aligned}
  h_{11} X_1 + \ldots + h_{1r} X_r + g_1 y_1 &= d'_1 \\
  &\vdots \\
  h_{l1} X_1 + \ldots + h_{lr} X_r + g_l y_l &= d'_l
\end{aligned}
$$

Hence, the linear reachability problem is reduced to finding a nonnegative integer solution to (12). Using (7) and (8), a simple calculation reveals that, in (12), all of the $h$'s are bounded by $|S| W_1$ and all of the $d'_1, \ldots, d'_l$ are bounded by $|S|^2 W_1 + W_2$.

We use $I_1$ to denote the maximum of the absolute values of all the $t \times t$, $1 \leq t \leq l$, minors of the coefficient matrix for system (12) and $I_2$ to denote that of the augmented matrix. Using the above mentioned bounds for the coefficients and constants in (12), one can conclude that

$$
I_1 \leq (|S| W_1)^l! \quad \text{and} \quad I_2 \leq (|S|^2 W_1 + W_2)(|S| W_1)^{l-1}!.
$$

A direct application of the Borosh-Flahive-Treybig bound in Theorem 1 shows that system (12) has solutions in nonnegative integers iff the system has a solution $(X_1, \ldots, X_r, y_1, \ldots, y_l)$ in nonnegative integers, among which $r$ unknowns are bounded by $I_1$ and $l$ unknowns are bounded by $(r+1) I_2$ (here, without loss of generality, we assume the worst case that the rank of coefficient matrix of (12) is $l$). Applying the bounds $I_1$ and $(r+1) I_2$ to $X_j$ in (5) and using (7) and (8), the linear reachability problem is further reduced to the problem of finding a $p \in P$ satisfying:

$$
\#(p) \leq (|S|^2 + (r - l)|S| I_1 + l|S|(r + 1) I_2) \cdot 1
$$

and $U(\#_{a_1}(p), \ldots, \#_{a_k}(p))$. Noticing that $l \leq L$, and $r \leq |S|^k$ according to (6), we apply the bounds of $I_1$ and $I_2$ in (13) to (14) and define a bounding box

$$
B = (|S|^{k+2} W_1 + L |S|(|S|^k + 1)(|S|^2 W_1 + W_2)(|S|W_1)^{L-1}L! + |S|^2).
$$
Hence,

**Theorem 3.** Given a finite state transition system \( A \), two states \( s_{\text{init}}, s_{\text{final}} \in S \), and a linear constraint \( U(x_1, \ldots, x_k) \), the following two items are equivalent:

- There is a path \( p \) of \( A \) from \( s_{\text{init}} \) to \( s_{\text{final}} \) satisfying \( U \),
- The above item is true for some \( p \) further satisfying \( \#(p) \leq B \cdot 1 \), where \( B \) is defined in (15).

Notice that \( B \) in (15) is independent of \( m \) in (10). Also, when the number of states \( |S| \) in \( A \) is much larger than \( k \) and the metrics of \( U \); i.e., \( |S| \gg k, W_1, W_2, L \), the bounding box is in the order of

\[
B = O(|S|^{k+L+3}). \tag{16}
\]

In this case, one can easily show that the linear reachability problem is solvable in time

\[
O(|S|^{2k(k+L+3)+2}). \tag{17}
\]

4 The Linear Liveness Problem

An \( \omega \)-path \( \pi \) of \( A \) is an infinite sequence such that each prefix is a path of \( A \). Let \( s \) and \( s' \) be any two designated states of \( A \). We say that \( \pi \) is \( U \)-i.o. (infinitely often) at \( s' \) if there are infinitely many prefixes \( p \) from \( s \) to \( s' \) of \( A \) such that \( p \) satisfies \( U \) (i.e., \( U(\#a_1(p), \ldots, \#a_k(p)) \) holds). The linear liveness problem can be formulated as follows:

- **Given:** A finite state transition system \( A \) in (2), two designated states \( s \) and \( s' \), and a linear constraint \( U(x_1, \ldots, x_k) \).
- **Question:** Is there an \( \omega \)-path \( \pi \) that starts from \( s \) and is \( U \)-i.o. at \( s' \)?

In [11], this problem is shown decidable. However, the time complexity is unknown. In this section, we reduce the liveness problem to a linear reachability problem.

Recall that \( U \) is in the form of (10), \( U = \lor_{1 \leq i \leq m} U_i \), and \( U_i^{\text{hom}} \) is the result of making \( U_i \) homogeneous. One key observation is as follows. The **Question**-part in the linear liveness problem is true iff, for some \( 1 \leq i \leq m \), \( (a) \), there is a path of \( A \) from \( s \) to \( s' \) satisfying \( U_i \), and, \( (b) \), there is a path of \( A \) from \( s' \) to \( s' \) satisfying \( U_i^{\text{hom}} \). A proof of this observation can be followed from [11] using the pigeon-hole principle (noticing that each atomic linear constraint in \( U_i \) is in the form of (4) where \( \sim \in \{=, \geq\} \)). Both items are equivalent to the linear reachability problem for \( A \) concerning \( U_i \) and \( U_i^{\text{hom}} \), respectively. By trying out all of the \( m \) number of \( U_i \)'s and \( U_i^{\text{hom}} \)'s, and using Theorem 3 and (17), we conclude that:

**Theorem 4.** The linear liveness problem is solvable in time shown in (17), when \( |S| \gg m, k, W_1, W_2, L \).
5 Ordered Finite State Transition Systems

Let $A$ be a finite state transition system in $(\mathcal{S}, \mathcal{A})$. Suppose that an order of labels $a_1, \cdots, a_k$ is fixed, say $a_1 < \cdots < a_k$. $A$ is ordered if, on any path $p$ from $s_{\text{init}}$ to $s_{\text{final}}$, each label $a_i$ appears before each label $a_j$, whenever $i < j$. In this case, $A$ behaves as follows: reading $a_1$’s for 0 or more times, then reading $a_2$’s for 0 or more times, and so on. For this restricted version of $A$, we will derive a better complexity bound than (17) for the linear reachability problem.

Lemma 2. The linear reachability problem for ordered $A$ is solvable in time

$$O(m \cdot |S|^{4k-2} \cdot P^{2k}),$$

where $P$ is the Pottier bound for $U$ (i.e., the maximum of the Pottier bounds for all $U_i$’s). Furthermore, since $P$ is independent of $|S|$, the linear reachability problem for ordered $A$ is solvable in time

$$O(|S|^{4k-1}),$$

when $|S| \gg m, k, P$.

Interestingly, this model of $A$ and the complexity bound can be used to obtain a complexity bound for timed automata in the next section.

6 The Linear Reachability Problem for Timed Automata

A timed automaton [1] is a finite state machine augmented with a number of clocks. All the clocks progress synchronously with rate 1, except when a clock is reset to 0 at some transition. We first consider discrete timed automata where clocks take integral values. Formally, a discrete timed automaton $D$ is a tuple

$$\langle S, \{x_1, \cdots, x_k\}, E \rangle,$$

where $S$ is a finite set of (control) states, $x_1, \cdots, x_k$ are clocks taking values in $\mathbb{N}$, and $E$ is a finite set of edges or transitions. Each edge $\langle s, \lambda, l, s' \rangle$ denotes a transition from state $s$ to state $s'$ with enabling condition $l$ in the form of clock regions (i.e., $x \# c, x - y \# c$, where $x, y$ are clocks, $\#$ denotes $\leq, \geq,$ or $=$, and $c$ is an integer) and a clock reset set $\lambda \subseteq \{1, \cdots, k\}$. Sometimes, we also write the edge as $s \xrightarrow{\lambda} s'$, or simply $s \xrightarrow{l} s'$ when $l$ is true. Without loss of generality, we assume that $|\lambda| \leq 1$. That is, each transition resets at most one clock. (Resetting several clocks can be simulated by resetting one by one.) When $\lambda = \emptyset$, the edge is called a progress transition. Otherwise, it is a reset transition. $D$ is static if the enabling condition on each edge is simply true.

The semantics of $D$ is defined as follows. A configuration is a tuple of a control state and clock values. Let $\langle s, v_1, \cdots, v_k \rangle$ and $\langle s', v'_1, \cdots, v'_k \rangle$ be two configurations. $\langle s, v_1, \cdots, v_k \rangle \rightarrow \langle s', v'_1, \cdots, v'_k \rangle$ denotes a one-step transition satisfying all of the following conditions:
There is an edge $\langle s, \lambda, l, s' \rangle$ in $A$.
- The enabling condition of the edge is satisfied; i.e., $l(v_1, \cdots, v_k)$ is true.
- If $\lambda = \emptyset$ (i.e., a progress transition), then every clock progresses by one time unit; i.e., $v'_i = v_i + 1$, $1 \leq i \leq k$. (iv). If for some $j$, $\lambda = \{j\}$ (i.e., a reset transition), then $x_j$ resets to 0 and all the other clocks do not change; i.e., $v'_j = 0$ and $v'_i = v_i$ for each $1 \leq i \neq j \leq k$.

We say that $\langle s, v_1, \cdots, v_k \rangle$ reaches $\langle s', v'_1, \cdots, v'_k \rangle$ if

$$\langle s, v_1, \cdots, v_k \rangle \rightarrow^* \langle s', v'_1, \cdots, v'_k \rangle,$$

where $\rightarrow^*$ is the transitive closure of $\rightarrow$.

The linear reachability problem for discrete timed automata is defined as follows.

- **Given:** A discrete timed automaton $D$, two designated states $s_{\text{init}}$ and $s_{\text{final}}$, and two linear constraints $U$ and $U'$ over $k$ variables.
- **Question:** are there clock values $v_1, \cdots, v_k, v'_1, \cdots, v'_k$ such that $\langle s_{\text{init}}, v_1, \cdots, v_k \rangle$ reaches $\langle s_{\text{final}}, v'_1, \cdots, v'_k \rangle$ and both $U(v_1, \cdots, v_k)$ and $U'(v'_1, \cdots, v'_k)$ hold?

The problem is decidable, even when clocks are dense. Its decidability proofs and application examples can be found in [8, 10, 9]. However, as we mentioned earlier, the time complexity for the problem is unknown. Using (18), we will obtain a complexity bound in this section.

Without loss of generality, we assume that both $U$ and $U'$ in the linear reachability problem for timed automata are a disjunction of $m$ conjunctive linear constraints. Each conjunctive linear constraint contains at most $L$ atomic linear constraints among which there are at most $E$ equations. Similar to Section 3, we use $W_1$ (resp. $W_2$) to represent the maximal value $\|B\|_{1, \infty}$ (resp. $\|b\|_{\infty}$) of all the conjunctive linear constraints $Bx \sim b$ in $U$ and $U'$. The complexity of the linear reachability problem will be measured on, among others, $L$, $E$, $m$, $W_1$, $W_2$, $|S|$, and $k$.

We first consider a simpler case when $D$ is static. Before we proceed further, more definitions are needed. A reset order $\tau$ is a sequence $\lambda_1, \cdots, \lambda_n$, for some $1 \leq n \leq k$, where each $\lambda_i$ contains exactly one element in $\{1, \cdots, k\}$, and all of the $\lambda_i$’s are pairwise disjoint. Let $\lambda_0 = \{1, \cdots, k\} - \cup_{1 \leq i \leq n} \lambda_i$. An execution path of $D$ is of reset order $\tau$ if every clock in $\lambda_0$ does not reset on p, and for rest of the clocks, their last resets are in this order: $x_{i_1}, \cdots, x_{i_n}$, with $\lambda_1 = \{i_1\}, \cdots, \lambda_n = \{i_n\}$. For the instance of the linear reachability problem of static $D$, we consider the **Question**-part witnessed by an execution path that is of any fixed reset order $\tau$ (there are only finitely many reset orders). From this instance and the given $\tau$, we will construct an ordered finite state transition system $A^\tau$ and a linear constraint $U^\tau$. Then, we reduce the linear reachability problem of $D$ to the linear reachability problem of $A^\tau$. The key idea behind the construction is as follows. Suppose $\lambda_0 \neq \emptyset$. The execution path can then be partitioned into $n + 1$ segments separated by the $n$ last resets given in $\tau$. We use $y_0, y_1, \cdots, y_n$ to
denote the number of progress transitions made on each segment respectively. Suppose that the path starts with clock values $z_1, \ldots, z_k$ and ends with clock values $x_1, \ldots, x_k$. Observe that each $x_i$ can be represented as a sum on some of $z_1, \ldots, z_k$ and $y_0, y_1, \ldots, y_n$. The case when $\lambda_0 = \emptyset$ is similar. Following this line, one can show:

**Lemma 3.** The linear reachability problem for static discrete timed automata $\mathcal{D}$ is solvable in time

$$O(k! \cdot m^2 \cdot (k + (k + 1) \cdot |S|)^{8k-2} \cdot (2 + k \cdot W_1 + W_2)^{(2k+2L+2E)-4k}).$$

Hence, when $|S| \gg k, m, W_1, W_2$, the linear reachability problem for static $\mathcal{D}$ is solvable in time

$$O(|S|^{8k-1}).$$

Now, we consider the case when $\mathcal{D}$ is not necessarily static. Let $C$ be one plus the maximal absolute value of all the constants appearing in enabling conditions in $\mathcal{D}$.

We use $T$ to denote the result of (20) after replacing $|S|$ with $(1 + 2C)^{k^2+k} \cdot |S|$, $L$ with $L + k$, $E$ with $E + k$, $W_1$ with $\max(W_1, 2)$, $W_2$ with $\max(W_2, C)$.

From [12, 10], one can construct a static $\mathcal{D}'$ with two designated states $s'_{\text{init}}$ and $s'_{\text{final}}$, and with at most $(1 + 2C)^{k^2+k} \cdot |S|$ number of states to simulate $\mathcal{D}$ faithfully. From this result, one can show, using Lemma 3.

**Theorem 5.** The linear reachability problem for discrete timed automata is solvable in time

$$O(k! \cdot (1 + C)^k \cdot T).$$

Again, when $|S|$ and $C$ are $\gg$ the size of $U$ and $U'$, the time complexity of linear reachability problem for discrete timed automata is

$$O(|S|^{8k-1} \cdot C^k+(k^2+k)\cdot(8k-2)+(6k+2L+2E)\cdot4k).$$

**Remark 2.** Using the techniques in Section 4, one can obtain a complexity bound similar to (23) for the linear liveness problem [12] for discrete timed automata. Also the complexity in (23) is more sensitive to $C$ than to $|S|$.

We turn now to the case when $\mathcal{D}$ is a timed automaton with $k$ dense clocks. One can similarly formulate the semantics and the linear reachability problem for $\mathcal{D}$ (e.g., see [9]). With the pattern technique presented in [9], it is easy to show the following. From $\mathcal{D}$ and $U, U'$, one can construct a discrete timed automaton $\mathcal{D}'$ with $k$ discrete clocks and two linear constraints $W, W'$ such that the linear reachability problem of timed automaton $\mathcal{D}$ concerning $U, U'$ is equivalent to the linear reachability problem of discrete timed automaton $\mathcal{D}'$ concerning $W, W'$. In addition, the number of states in $\mathcal{D}'$ is $O(2^{6(k+1)^2} \cdot |S|)$, where $S$ is the state set in $\mathcal{D}$. (There are at most $2^{6(k+1)^2}$ patterns [9].) Furthermore, $W$ and $W'$ only depend on $U, U'$ and $k$ (independent of $\mathcal{D}$). Hence, we have the following conclusion:

**Theorem 6.** The linear reachability problem for timed automata with dense clocks can still be solvable in time shown in (23), when $|S|, C \gg k$ and the size of $U$. 

**Note:** The complexity in (23) is more sensitive to $C$ than to $|S|$. 


7 Conclusions

We obtained a number of new complexity results for various linear counting problems (reachability and liveness) for (ordered) finite state transition systems and timed automata. At the heart of the proofs, we used some known results in estimating the upper bound for minimal solutions (in nonnegative integers) for linear Diophantine systems. In particular, when all the parameters (such as the number of labels/clocks, the largest constant $C$ in a timed automaton, the size of the linear constraint to be verified, etc) except the number of states $|S|$ of the underlying transition system are considered constants, all of the complexity bounds obtained in this paper is polynomial in $|S|$. This is, as we mentioned in Section 1, in contrast to the exponential bounds that were previously known. In practice, a requirement specification (e.g., the $U$ in a linear counting problem) is usually small and simple $[14]$. In this sense, our results are meaningful, since the large size of $|S|$ is usually the dominant factor in efficiently solving these verification problems.

The counts of labels in a finite state transition system can be regarded as monotonic counters. Therefore, our result also provides a technique to verify safety properties for a special class of counter machines $M$ with monotonic counters: starting from a state $s$, whether it is always true that the counter values in $M$ satisfy a given linear constraint whenever $M$ reaches $s'$. In the future, we will study whether the techniques in this paper can be generalized to handle the case when $M$ is further augmented with an unrestricted counter (i.e., can be incremented, decremented, and tested against 0) or even a pushdown stack. Additionally, it is also desirable to study whether our techniques can be further generalized to the reachability problem of some classes of Petri nets $[18]$.

The authors would like to thank P. San Pietro and the anonymous referees for many valuable suggestions.

References


TCTL Inevitability Analysis of Dense-Time Systems

Farn Wang¹, Geng-Dian Hwang², and Fang Yu²

¹ Dept. of Electrical Engineering, National Taiwan University
Taipei, Taiwan 106, Republic of China
+886-2-23635251 ext. 435; FAX +886-2-23671909
farn@cc.ee.ntu.edu.tw

² Institute of Information Science, Academia Sinica
Taipei, Taiwan 115, Republic of China

Abstract. Inevitability properties in branching temporal logics are of the syntax \( \forall \diamond \phi \), where \( \phi \) is an arbitrary (timed) CTL formula. Such inevitability properties in dense-time logics can be analyzed with greatest fixpoint calculation. We present algorithms to model-check inevitability properties both with and without non-Zeno computation requirement. We discuss a technique for early decision on greatest fixpoint calculation. Our algorithms come with a \( d \)-parameter for the measurement of time-progress. We have experimented with various issues, which may affect the performance of TCTL inevitability analysis. Specifically, we report the performance of our implementation w.r.t. various \( d \)-parameter values and with or without the non-Zeno computation requirement in the evaluation of greatest fixpoints. We have also experimented with safe abstractions techniques for model-checking TCTL inevitability properties. Analysis of experiment data helps clarify how various techniques can be used to improve verification of inevitability properties.

Keywords: Branching temporal logics, TCTL, real-time systems, inevitability, model-checking, greatest fixpoint, abstraction

1 Introduction

In the research of verification, two types of specification properties often attract much interest from academia and industry. The first type specifies that "bad things will never happen" while the second type specifies that "good things will happen"[3]. In linear temporal logics, the former is captured by modal operator \( \Box \) and the latter by \( \Diamond [21] \). Tremendous research effort has been devoted to efficient analysis of these two types of properties in the framework of linear temporal logics[17]. In the branching temporal logics of (timed) CTL[1, 7], these
two concepts can be mapped to modal operators $\forall$ and $\forall\diamond$ respectively. $\forall$ properties are called safety properties while $\forall\diamond$ properties are usually called inevitability properties\cite{11, 19}. In the domain of dense-time system verification, people have focused on the efficient analysis of safety properties\cite{10, 13, 16, 20, 24, 25, 26, 27, 28, 31}. Inevitability properties in Timed CTL (TCTL)\cite{1, 12} are comparatively more difficult to analyze. In the framework of model-checking, to analyze an inevitability property, say $\forall\diamond\phi$, we actually compute the set of states that satisfy the negation of the inevitability, in symbols $\exists\Box\neg\phi$. We then use the intersection emptiness between $\exists\Box\neg\phi$ and the initial states for the answer to the inevitability anlaysis. However, property $\exists\Box\neg\phi$ in TCTL semantics is only satisfied with non-Zeno computations\cite{1}. Zeno computations are those counter-intuitive infinite computations whose execution times converge to a finite value\cite{12}. For example, a specification like

"along all computations, eventually a bus collision will happen in three time units"

can be violated by a Zeno computation whose execution time converges to a finite timepoint, e.g. 2.9. Such a non-Zeno requirement on computations may add complexity to the evaluation of inevitability properties. In this work, we present a symbolic TCTL model-checking algorithm which can handle the non-Zeno requirement in the evaluation of greatest fixpoints.

To contain the complexity of TCTL inevitability analysis, it is important to integrate new and old techniques for a performance solution. We investigate three approaches. In the first approach, we investigate how to adjust a parameter value in our greatest fixpoint evaluation algorithms for better performance. After the experiment reported in subsection 8.2, we shall suggest a strategy in choosing this parameter value for verification performance.

In the second approach, we present a technique called Early Decision on the Greatest Fixpoint (EDGF). The idea is that, in the evaluation of the greatest fixpoints, we start with a state-space and iteratively pare states from it until we reach a fixpoint. Thus, if in a middle greatest fixpoint evaluation iteration, target states have already been pared from the greatest fixpoint, we can conclude that these target states are not included in the fixpoint. Through this technique, we can reduce time-complexity irrelevant to the answer of the model-checking.

Our third approach is to use abstraction techniques\cite{6, 30}. We shall focus on a special subclass TCTL$^\forall$ of TCTL in which every formula can be analyzed with safe abstraction if over-approximation is used in the evaluation of its negation. For model-checking formulas in TCTL$^\forall$, abstraction accuracy can be a big issue because the inaccuracy in abstraction can potentially be magnified when we use inaccurate evaluation results of nested modal subformulas to evaluate nesting modal subformulas using abstraction techniques. Thus it is important to discern the accuracy of previous abstraction techniques to discern true TCTL$^\forall$ formulas.

In this paper, we also discuss another possibility for abstract evaluation of greatest fixpoints, which is to omit the requirement for non-Zeno computations in TCTL semantics. As reported in section 8, many benchmarks are true even without exclusion of Zeno computations.
We have implemented these ideas in our model-checker/simulator \texttt{red} 4.1[24, 25, 26]. We report our experiments to observe the effects of our proposed techniques on inevitability analysis. We also compare our analysis with Kronos 5.1[31], which is a model-checker for full TCTL. The performance data shows good promise of our techniques. We envision that our techniques can be valuable components in the arsenal of verification technology. Engineers can select proper combinations of weapons from the arsenal to complete their verification tasks.

Engineers may feel clueless in how to find a performance combination of the techniques reported in this paper (or in the literature). After all, if we have to run the verification tasks to their completions to check the performance of a combination, then the techniques presented here are really not justified. Finally, we present a predictive method to tell us how a combination of techniques perform without having to run verification tasks to completion with this combination.

2 Related Work

The TA model with dense-time clocks was first presented in \cite{2}. Notably, the data-structure of DBM is proposed in \cite{9} for the representation of convex state-spaces of TA. The theory and algorithm of TCTL model-checking were first given in \cite{1}. The algorithm is based on region graphs and helps manifest the PSPACE-complexity of the TCTL model-checking problem.

In \cite{12}, Henzinger et al. proposed an efficient symbolic model-checking algorithm for TCTL. Our greatest fixpoint evaluation algorithm extends it to handle non-Zeno requirement in general inevitability analysis. So far, several verification tools for TA have been devised and implemented\cite{13, 16, 20, 24, 25, 26, 27, 28, 29, 31}. UPPAAL\cite{20} is a popular tool using DBM technology. Recently, Moller has used UPPAAL with abstraction techniques to analyze restricted inevitability properties with no modal-formula nesting\cite{18}. The idea is to make model augmentations to speed up the verification performance. Moller also shows how to extend the idea to analyze TCTL with only universal quantifications. However, no experiment has been reported on the verification of nested modal-formulas.

Kronos\cite{31} is a full TCTL model-checker using DBM technology with both forward and backward reasoning capability. Experiments to demonstrate the use of Kronos to verify several TCTL bounded inevitability (inevitabilities with specified deadline) properties is demonstrated in \cite{31}. No report has been made on how to enhance the performance of general inevitability analysis. In comparison, we discuss techniques, like EDGF and abstractions, which handle both bounded and unbounded inevitabilities.

Our tool \texttt{red} (version 4.1)\cite{26} is a full TCTL model-checker/simulator with a BDD-like data-structure, called CRD (clock-restriction diagram)\cite{24, 25, 26}. Previous research with \texttt{red} has focused on enhancing the performance of safety analysis\cite{22, 23, 24, 25, 26}.

Abstraction techniques for safety analysis have been studied in great depth since the pioneer work of Cousot et al\cite{6}. For TA, convex-hull over-approximation\cite{30} has been a popular choice for DBM technology. It is difficult
to implement this over-approximation in red[26] since variable-accessing has
to observe variable-orderings of BDD-like data-structures. Nevertheless, many
over-approximation techniques for TA have been reported in [4] for BDD-like
data-structures and in [29] specifically for CRD.

Relationships between abstraction techniques and subclasses of CTL with
only universal (or existential respectively) path quantifiers has been studied
in [8]. As mentioned, the corresponding framework in TCTL is noted in [18].

3 TCTL Model-Checking
We use TCTL model-checking as our verification framework, in which we are
given a timed automata (TA)[2] as behavior description and a TCTL formula[1]
as specification and want to check whether the behavior description satisfies
the specification. A TA is a finite-state automata equipped with a finite set of
clocks that can hold nonnegative real-values. In its operation, one transition can
be triggered when a corresponding triggering condition is satisfied. Upon being
triggered, the automata instantaneously transits from one mode to another and
resets some clocks to zero. Between transitions, all clocks increase readings at
a uniform rate.

TCTL[1, 12] is a branching temporal logic for the specification of dense-time
 systems. A TCTL formula $\phi$ has the following syntax rules.

$$\phi ::= q | x \leq c | \neg \phi_1 | \phi_1 \lor \phi_2 | x.\phi_1 | \exists \phi_1 U \phi_2 | \exists \phi_1$$

Here, $q$ is mode name and specifies the current mode, $x \leq c$ is a timing constraint
that compares the reading of a clock ($x$) with a contant $c$, and $\phi_1, \phi_2$ are TCTL
formulas. Due to page-limit, we will not go into details. For details, see [1, 12].

4 Model-Checking Algorithm
with Non-Zeno Requirements
Our model-checking algorithm is backward reasoning. We need two basic pro-
cedures, $\text{transition bck}()$ for the computation of the weakest precondition of transi-
tions, and $\text{time bck}()$ for that of backward time-progression. With these two ba-
sic procedures, we can construct a backward reachability procedure
$\text{reachable-bck}(\eta_1, \eta_2)$, as shown in [12, 22, 23, 24, 25, 26, 28], which char-
acterizes the backwardly reachable state-space from states in $\eta_2$ through runs
along which all states satisfy $\eta_1$. $\text{reachable-bck}(\eta_1, \eta_2)$ can be defined as

$$\text{reachable-bck}(\eta_1, \eta_2) \equiv \text{lfp } Y.(\eta_2 \lor (\eta_1 \land \text{time bck}(\eta_1 \land \forall e \in T. \text{transition bck}(Y, e))))$$

Our model-checking algorithm extends from the classic model-checking al-
gorithm for TCTL[12]. The design of our greatest fixpoint evaluation algorithm
with consideration of non-Zeno requirement is based on the following lemma.

Lemma 1. Given $d \geq 1$, $A, \nu \models \exists \square \eta$ iff there is a finite run from $\nu$ of duration
$\geq d$ such that along the run every state satisfies $\eta$ and the finite run ends at
a state satisfying $\exists \square \eta$. 
**Proof:** Details are omitted due to page-limit. Note that we can construct an infinite and divergent run by concatenating an infinite sequence of finite runs with durations \( d \geq 1 \). The existence of infinitely many such concatenateable finite runs is assured by the recursive construction of \( \exists \Box \eta \). 

Then, \( \exists \Box \eta \) can be defined with the following greatest fixpoint

\[
\exists \Box \eta \equiv \text{gfp } Y.(ZC.\text{reachable-bck}(\eta, Y \land ZC \geq d))
\]

Here clock \( ZC \) is used specifically to measure the non-Zeno requirement. The following procedure can construct the greatest fixpoint satisfying \( \exists \Box \eta \) with a non-Zeno requirement.

**Procedure gfp()** /* \( d \) is a static parameter for measuring time-progress */ {
\[\]
Y := \eta; \quad Y' := \text{true};
repeat until \( Y = Y' \),
\{ Y' := Y; \quad Y := Y \land \text{clock\_eliminate}(ZC = 0 \land \text{reachable-bck}(\eta, Y \land ZC \geq d), ZC); \}
return \( Y \);
\]

Here \( \text{clock\_eliminate()} \) removes a clock from a state-predicate (Boolean combination of atoms like \( q \) or \( x \leq c \)) without losing information on relations among other clocks. Note here that \( d \) works as a parameter. We can choose the value of \( d \geq 1 \) for better performance in the computation of the greatest fixpoint.

Procedure \( \text{gfp()} \) can be used in the labeling algorithm in [1, 12] to replace the evaluation of \( \exists \Box \)-formulas. Due to page-limit, we omit the complete model-checking algorithm. Details can be found in a full-version of this report at http://arxiv.org/abs/cs.SC/0304003. Correctness follows from Lemma 1.

## 5 Early Decision on Greatest Fixpoint Evaluation

In the evaluation of the greatest fixpoint for formulas like \( \exists \Box \phi_1 \), we start from the description, say \( Y \), for a subspace of \( \phi_1 \). We iteratively eliminate subspaces which cannot go to a state in \( Y \) through finite runs of \( d \geq 1 \) time units. Thus, the state-space represented by \( Y \) shrinks iteratively until it settles at a fixpoint. In practice, this greatest fixpoint usually happens in conjunction with other formulas. For example, we may want to specify \( \text{collision} \rightarrow y.\forall(\neg(y < 26 \land \text{idle}) \) meaning that a bus at the collision state will enter the idle state within 26 time-units. After negation for model-checking, we get \( \text{collision} \land y.\exists(y \geq 26 \lor \neg\text{idle}) \). In evaluating this negated formula, we want to see if the greatest fixpoint for the \( \exists \Box \)-formula intersects with the state-space for \( \text{collision} \). We do not actually have to compute the greatest fixpoint to know if the intersection is empty. Since the value of \( Y \) iteratively shrinks, we can check if the intersection between \( Y \) and the state-space for \( \text{collision} \) becomes empty at each iteration of the greatest fixpoint construction (i.e., the repeat-loop at statement (1) in procedure \( \text{gfp()} \)). If at an iteration we find the intersection with \( Y \) is already empty, there is no need to continue calculating the greatest fixpoint and we can immediately return the current value of \( Y \) (or \( \text{false} \)) without affecting the result of model-checking.
6 Greatest Fixpoint Computation by Tolerating Zenoness

In practice, greatest fixpoint computation procedures presented in the last two sections can be costly in computing resources since their characterizations have a least fixpoint nested in a greatest fixpoint. This is necessary to guarantee that only non-Zeno computations are considered. In reality, it may happen that, due to well-designed behaviors, systems may still satisfy certain inevitability properties for both Zeno and non-Zeno computations. In some cases, we can benefit from the following less expensive procedure to compute the greatest fixpoint.

$$\exists \Box \eta \equiv \text{gfp } Y. (\eta \land \text{time}_bck(\eta \land \lor_{e \in T \text{xtion}_bck(Y,e)}))$$

Even if the procedure can over-estimate the greatest fixpoint, it can be much less expensive in the verification of well-designed real-world projects.

7 Abstract Model-Checking with TCTL$^\forall$

In the application of abstraction techniques, it is important to make them safe[30]. That is to say, when the safe abstraction analyzer says a property is true, the property is indeed true. When it says false, we do not know whether the property is true. There are two types of abstractions: over-approximation, which means that the abstract state-space is a superset of the concrete state-space, and under-approximation, which means that the abstract state-space is a subset of the concrete state-space. To make an abstraction safe, we should over-approximate when evaluating $$\exists \Box \neg \phi$$ (the negation of the inevitability). However, negations deeply nested in formulas can turn over-approximation into under-approximation and thus make abstractions unsafe.

To guarantee safe abstraction in model-checking, in the literature[8, 18], people focused on subclass TCTL$^\forall$ of TCTL, in which every formula can be analyzed with safe abstraction when using over-approximation. A formula is in TCTL$^\forall$ iff the negation signs only appear before its atoms, and only universal path quantifications are used. For example, we may write formula $$\forall \Box (\text{request} \rightarrow \forall \Box (\text{service} \rightarrow \forall \Box \text{request}))$$. This subclass allows for nested modal formulas and captures many TCTL inevitability properties.

We can extend reachable-bck() with over-approximation as follows.

$$\text{reachable-bck}^O(\eta_1, \eta_2) \equiv \text{lfp } Y. \text{abs } ((\eta_2 \lor (\eta_1 \land \text{time}_bck(\eta_1 \land \lor_{e \in T \text{xtion}_bck(Y,e)}))))$$

Here, abs() means a generic over-approximation procedure. In our tool red 4.1, we have implemented a series of game-based abstraction procedures. Details can be found in [29].

8 Experiments

We design our experiment in two ways. First, we run red 4.1 with various options and benchmarks to test if our ideas can indeed improve the verification performance of inevitability properties in TCTL$^\forall$. Second, we compare red 4.1 with
Kronos 5.2 to check whether our implementation is competitive. Note that comparison result should be read carefully since red uses BDD-like data-structures while Kronos uses DBM. It is difficult to conclude if the techniques presented in this work definitely contribute to the performance difference between red and Kronos. Nevertheless, we believe it is still an objective measure to roughly estimate how our ideas perform.

8.1 Benchmarks

Benchmarks include leader-election protocol[26], PATHOS real-time operating system scheduling specification[4], and CSMA/CD protocol[24, 25, 31]. We use unbounded inevitability specifications for election-election, PATHOS, and CSMA/CD (C), and bounded inevitability specifications[12] for CSMA/CD (A) and CSMA/CD (B). The nesting depth of the modal operators are zero for leader-election, one for PATHOS, CSMA/CD (A), and CSMA/CD (B), and two for CSMA/CD (C).

8.2 Performance w.r.t. Parameter for Measuring Time-Progress

In statement (2) of procedure gfp(), we use inequality \( ZC \geq d \) to check time-progress in non-Zeno computations where \( d \) is a parameter \( \geq 1 \). In our experiment, we want to learn how to choose the value of parameter \( d \) for verification performance. Various values of parameter \( d \) are used, ranging from 1 to beyond the biggest timing constants used in the system models. For the leader-election benchmark, the biggest timing constant used is 2. For the Pathos benchmark, it is equal to the number of processes. For the CSMA/CD benchmarks (A), (B), and (C), it is 808.

We have drawn charts to show time-complexity for the benchmarks w.r.t. \( d \)-values in figure ???. More performance data can be found in our full-version paper.

As can be seen from the charts, our algorithms may respond to various model structures and specifications with different complexity curves. For benchmarks leader-election and PATHOS, the bigger the \( d \)-value, the better the performance. For the three CSMA/CD benchmarks, the best performance happens when \( d \) is around 80. One common attribute in these charts is that \( d = 1 \) always gives the worst performance.

We have looked into the execution of our algorithms for explanation of the complexity curves. Procedure gfp() is constructed with an inner loop for the least fixpoint evaluation of reachable-bck() and an outer loop for the greatest fixpoint evaluation. The outer loop converges faster with bigger \( d \)-values while the inner loop converges slower. With bigger \( d \)-values we may need less iterations of the outer-loop and, at the same time, more iterations of the inner loop to compute the greatest fixpoints. The complexity patterns in the charts are thus superpositions between the complexities of the outer loop and the inner loop.
**SUGGESTIONS ON HOW TO CHOOSE \(d\)-values:** First, small values around \(d = 1\) should be avoided. Second, since the curves are in general smooth and non-monotonic beyond the range of small \(d\)-values, engineers may want to check how fast the greatest fixpoint iterations proceed on a few bigger \(d\)-values to determine the monotonicity of the curve for bigger \(d\)-values. After we have established the monotonicity of a verification target, then we can choose a better efficient \(d\)-value among the bigger \(d\)-values.

We use the \(d\)-values that achieve the best performance for our experiments reported in the next few subsections.

8.3 Performance w.r.t. Non-Zeno Requirement and EDGF

In this experiment, we observe the performance of our algorithm w.r.t. the non-Zeno requirement and the EDGF policy. Performance data is shown in table ??.

In general, we find that with or without the EDGF technique, a non-Zeno requirement does add more complexity to the evaluation of the inevitability properties. For the three specifications of the CSMA/CD model, exponential blow-ups were observed. Conversely, for the PATHOS benchmark, the non-Zeno requirement seems to incur much less complexity than without it. After we have carefully traced the execution of our mode-checker, we found that this benchmark incurs very few iterations of outer loop with non-Zeno requirement although each iteration can be costly to run. On the other hand, it incurs a significant number of iterations of inner loop without non-Zeno requirement although each iteration is not so costly. The accumulative effect of the loop iterations result in performance that contradicts our expectation. This benchmark shows that the evaluation performance of inevitability properties is very involved and depends on many factors. Furthermore, benchmark CSMA/CD (B) shows that some inevitability properties can only be verified with non-Zeno computations.

As for the performance of the EDGF technique, we find that when the technique fails, it only incurs a small overhead. When it succeeds, it significantly improves performance two to three-fold.

8.4 Performance w.r.t. Abstraction Techniques

In table ??, we report the performance data of our red 4.1 with respect to our three abstraction techniques.

In general, the abstraction techniques give us much better performance. Notably, game-discrete and game-magnitude abstractions seem to have enough accuracy to discern true properties.

It is somewhat surprising that the game-magnitude abstraction incurs excessive complexity for PATHOS benchmark. After carefully examining the traces generated by red, we found that because non-magnitude constraints were eliminated, some of the inconsistent convex state-spaces in the representation became consistent. These spurious convex state-spaces make many more paths in our CRD and greatly burden our greatest fixpoint calculation. For instance, the
outer loop of procedure $\text{gfp()}$ takes two iterations to reach the fixpoint with the game-magnitude abstraction. It only takes one iteration to do so without the abstraction. In our previous experience, this abstraction has worked efficiently with reachability analysis. It seems that the performance of abstraction techniques for greatest fixpoint evaluation can be subtle.

8.5 Performance w.r.t. Kronos

In table ??, we report the performance of Kronos 5.2 w.r.t. the five benchmarks. For PATHOS and leader election, Kronos did not succeed in constructing the quotient automata. But our red seems to have no problem in this regard with its on-the-fly exploration of the state-space. Of course, the lack of high-level data-variables in Kronos’ modeling language may also acerbate the problem.

As for benchmark CSMA/CD (A), Kronos performs very well. We believe this is because this benchmark uses a bounded inevitability specification. Such properties have already been studied in the literature of Kronos[31].

On the other hand, benchmarks CSMA/CD (B) and (C) use unbounded inevitability specifications with modal-subformula nesting depths 1 and 2 respectively. Kronos does not scale up to the complexity of concurrency for these two benchmarks. Our red prevails in these two benchmarks.

9 A Strategy to Combine the Techniques

We here present a predictive method to tell us how a combination of techniques perform without having to run verification tasks to completion with this combination. Real-world verification tasks usually takes many fixpoint iterations to complete. Given a combination of techniques, the first few iterations usually cost little computer resources but may still give accurate prediction on how the combination performs. Thus we suggest verification engineers to choose their combination of techniques based on collected data on execution times and memory sizes consumed by the first few iterations of various combinations.

References


Abstract. This paper focuses on the realizability problem of a framework for modeling and specifying the global behavior of reactive electronic services (e-services). In this framework, Web accessible programs (peers) communicate by asynchronous message passing, and a virtual global watcher listens silently to the network. The global behavior is characterized by a conversation, which is the infinite sequence of messages observed by the watcher. We show that given a Büchi automaton specifying the desired set of conversations, called a conversation protocol, it is possible to implement it using a set of finite state peers if three realizability conditions are satisfied. In particular, the synthesized peers will conform to the protocol by generating only those conversations specified by the protocol. Our results enable a top-down verification strategy where: (1) A conversation protocol is specified by a realizable Büchi automaton, (2) The properties of the protocol are verified on the Büchi automaton specification, (3) The peer implementations are synthesized from the protocol via projection.

1 Introduction

The use of e-services (i.e., self-contained Web accessible programs and devices) has revolutionized the way that business services are provided and deployed. One recent trend is to provide value added composite e-services by integrating existing services available on web. However to make such a “composite paradigm” prevail, one has to first resolve the modeling problem, i.e., how to define the public invocation interface so that individual e-services can be discovered and invoked by others (see [18]).

It has been realized that stateless function call models like WSDL [29] are not adequate to describe long running complex services. Indeed [15] shows that lack of stateful coordination of server scripts caused problems in Orbitz reservation, and similar flaws are also observed in many other services like Hertz Rental and Register.com. Emerging standards like BPML [7], BPEL4WS [6] and WSCI [28] can resolve this problem by exposing the abstract control flow skeleton of business processes so that invoker knows how to interact. In contrast to the process
oriented view of BPEL4WS [6], IBM Conversation Support Project [16] concentrates on the interaction in a peer-to-peer conversation session, and proposes the notion of conversation policy. In [9] we generalized the idea of conversation policy to conversation specification (protocol), which describes conversation logic globally for any number of peers. A conversation protocol can be conveniently captured by a finite state automaton, with the set of messages exchanged among peers as the alphabet. A reactive version for e-services would simply use a Büchi automaton, which is a successful methodology for expressing liveness requirements.

Though increasingly many e-service standards [29, 6, 19] have been and are being proposed by the industry, many fundamental issues remain unclear [18]. For example, one issue is what the underlying communication model for e-services should be. There has been extensive work on synchronous communication models, for example, CSP [17], I/O automata [22] and interface automata [3]. However, their synchronous assumption, i.e., two communicating processes should execute a send and a corresponding receive action synchronously, is not realistic in an environment like Internet, where there is no global clock and network delay is significant. Although asynchrony can be simulated by introducing explicit queue processes in synchronous model, like [14], the introduction of queue processes inhibits the direct application of finite state model checking tools.

In our previous work [9], a framework was developed for modeling e-services with asynchrony assumptions. Under this framework, peers (individual e-service components) communicate via asynchronous messages and each peer maintains a queue for incoming messages. A virtual global watcher keeps track of messages as they occur, i.e., each sent message is simultaneously written to an input queue and concatenated to the watcher. A central notion of a conversation, which is a finite sequence of messages observed by the watcher, was studied. This model can be regarded as a theoretical abstraction of many industry efforts like Java Message Service [27].

Continuing on the study of e-service conversations, this paper extends the model in [9] by focusing on reactive e-services, where the global conversation is always infinite (However, some peers may terminate in the composition). Similar to the results of [9] on finite words, in this paper we show that composition of finite state peers generates non ω-regular languages. In addition, we show that the problem of checking if the composition of finite state peers satisfies an LTL property is undecidable due to the unbounded input queues associated with peers. This motivates our top-down approach to specification of composite e-services. We specify the desired set of global conversations of an e-service using a Büchi automaton, and we call it a conversation protocol.

Unfortunately, not all conversation protocols are realizable. We present three realizability conditions in this paper and show that any conversation protocol which satisfies these three properties is realizable. The first property is called lossless join property which requires that the protocol should be complete – when projected to individual peers, the Cartesian product of the projections should be
exactly the same as the original protocol. The second property is the \textit{synchronous compatible} property which ensures that the protocol does not have “illegal states” as specified in \cite{3}. Finally the third condition is the \textit{autonomous property} which implies that at any point in the execution, each peer, independently, can make a decision on whether to send, or to wait for a message, or to terminate. LTL properties verified on a realizable conversation protocol will be preserved by its synthesized peers, and this result supports a top-down verification strategy.

\textbf{Related Work} Our model of composite e-services is different than the Communicating Finite State Machines (CFSM) model in \cite{8}. In our model message are exchanged through a virtual common medium and stored in the queue associated with the receiver, whereas in \cite{8} each pair of communicating machines use isolated communication channels and each channel has its own queue. The idea of using CFSM with FIFO queues to capture indefinite delay of messages (signals) is similar to many other published models like Codesign Finite State Machine \cite{10}, and Kahn Process Networks \cite{20}. Other formalisms like $\pi$-Calculus \cite{23} and the recent Microsoft Behave! Project \cite{26} are used to describe concurrent, mobile and asynchronously communicating processes.

Brand and Zafiropulo have shown in \cite{8} that CFSM with perfect FIFO queues are as powerful as Turing Machines. Thus it is not hard to infer that LTL model checking on our e-service composition model is undecidable. This undecidability result is caused by the unbounded FIFO queues, and in \cite{13}, many problems are proved to be undecidable even for two identical communicating processes. The transaction sequential consistency problem in \cite{5} provides another perspective for understanding the queue effect, where independent transactions are allowed to commute (which resembles our Prepone operator in \cite{9}). In \cite{2} it is shown that, if perfect FIFO channels are replaced by lossy channels, many problems become decidable. However we stick with the perfect FIFO in our model, since we assume that underlying communication protocols (like TCP/IP) ensure perfect FIFO message deliveries.

To the best of our knowledge, the notion of realizability on open/concurrent systems was first studied in the late 80’s (see \cite{1, 24, 25}). In \cite{1, 24, 25}, realizability problem is defined as whether a peer has a strategy to cope with the environment no matter how the environment decides to move. The concept of realizability studied in this paper is rather different. In our model the environment of an individual peer consists of other peers whose behaviors are also governed by portions of the protocol relevant to them. A related notion is the realizability of Message Sequence Chart (MSC) Graphs \cite{4}. However, the MSC Graph model captures both “send” and “receive” events, while in our e-composition model we are interested in the ordering of “send” events only. It can be shown that the MSC Graph model and our conversation protocol model are incomparable in expressiveness. In addition, the three realizability conditions proposed in \cite{4} are different than ours. For example, a conversation protocol which defines the language $m^\omega$ does not satisfy the \textit{bounded} condition of \cite{4}, but it satisfies the realizability conditions in this paper. It is interesting to note that there are other
works like fair reachability analysis in [21] which achieved decidable analysis results by restricting both the shape of composition (cyclic connection in [21]) and the control flow of protocol itself.

This paper is organized as follows. §2 defines the e-service model and in particular the notion of an e-service conversation. §3 discusses LTL model checking of e-services. §4 defines the concept of a conversation protocol and establishes the main results on the three realizability conditions. Finally §5 presents our conclusions.

2 A Model for E-services

We introduce the formal model of composite e-services in this section. In our model (see Fig. 1), an e-service consists of a set of peers where each peer maintains one input queue for incoming messages and may send messages to the queues of other peers. A global watcher listens silently to the network and observes the global behavior as a sequence of messages at the times of being sent (i.e. enqueued). The time a peer actually consumes a message from its queue is a local decision by the peer. Such a modeling of the global behavior of an e-service (or a distributed system) departs from the traditional approach of focusing on the behaviors of individual peers (or subsystems). In this section, we start with the modeling of individual peers. Then we move to the definition of composite e-services. Finally we introduce the notion of global configuration, based on which the concept of a conversation is defined.

For an alphabet $\Gamma$, let $\Gamma^*, \Gamma^\omega$ be the set of all finite, infinite (resp.) words over $\Gamma$, and $\Gamma^{\leq \omega} = \Gamma^* \cup \Gamma^\omega$. The definition of a peer is presented as follows.

**Definition 1.** A peer is modeled using a nondeterministic Büchi automaton $(\Sigma_{in}, \Sigma_{out}, T, s, F, \Delta)$ where (1) $\Sigma_{in}$ and $\Sigma_{out}$ are disjoint finite sets of incoming and outgoing (resp.) messages, (2) $T$ is a finite set of states, $s \in T$ is the initial state, and $F \subseteq T$ is a set of final states, (3) $\Delta \subseteq T \times (\Sigma_{in} \cup \Sigma_{out} \cup \{\epsilon\}) \times T$ is the transition relation. ($\epsilon$ is the empty word.)

A transition is either an $\epsilon$-move, or it either consumes an incoming message (from the input queue) or produces an output but not both. These three types of moves are denoted by triples $(q_1, \epsilon, q_2)$, $(q_1, ?a, q_2)$, $(q_1, !b, q_3)$ respectively. Let $L(p)$ represent the language accepted by a peer $p$. (A word is accepted if some final states are visited infinitely often.) Due to the presence of $\epsilon$-moves in a peer $p$, $L(p)$ may contain finite words, i.e., $L(p) \subseteq \Sigma^{\leq \omega}$ where $\Sigma = \Sigma_{in} \cup \Sigma_{out}$.

In the following we define the notion of an “e-service” which involves a set of peers. For convenience, we use $p_i$ to denote a peer, and the alphabets, the set of states, etc. of the peer $p_i$ also have subscript $i$. 

![Fig. 1. e-service model](image-url)
Definition 2. An e-service is a triple \((n, P, \sigma)\) where

- \(n > 1\) is an integer,
- \(P = \{p_1, p_2, \ldots, p_n\}\) is a set of \(n\) peers with pairwise disjoint input alphabets \(\Sigma_{i}^{\text{in}}, \ldots, \Sigma_{n}^{\text{in}}\) and pairwise disjoint output alphabets \(\Sigma_{i}^{\text{out}}, \ldots, \Sigma_{n}^{\text{out}}\) (note that \(\Sigma_{i}^{\text{in}} \cap \Sigma_{j}^{\text{out}}\) may be nonempty for \(i \neq j\)) such that \(\bigcup_{i} \Sigma_{i}^{\text{in}} = \bigcup_{i} \Sigma_{i}^{\text{out}}\), and
- \(\sigma: [1..n] \times (\bigcup_{i} \Sigma_{i}^{\text{out}}) \rightarrow [1..n]\) is a partial mapping such that
  - For each \(i \in [1..n]\) and each \(b \in \Sigma_{i}^{\text{out}}\), \(\sigma(i,b)\) is defined, and
  - For each \(i \in [1..n]\), each \(j \in [1..n]\), and each \(b \in \Sigma_{i}^{\text{out}} \cap \Sigma_{j}^{\text{in}}\), \(\sigma(i,b) = j\).

Intuitively, \(\sigma(i,a) = j\) means that the message \(a\) can be sent by peer \(p_i\) to peer \(p_j\). Note that in Definition 2 a message can be transmitted on one channel only. In the remainder of the paper, we use \(\Sigma\) to denote the entire set of alphabet of an e-service \((n, P, \sigma)\), i.e., \(\Sigma = \bigcup_{i} \Sigma_{i}^{\text{in}}\).

Definition 3. Let \(S = (n, \{p_1, \ldots, p_n\}, \sigma)\) be an e-service. A configuration of \(S\) is a \((2n+1)\)-tuple of the form \((Q_1, t_1, \ldots, Q_n, t_n, w)\) where for each \(j \in [1..n]\), \(Q_j \in (\Sigma_{j}^{\text{in}})^*\) is the queue content of peer \(p_j\), \(t_j\) is the state of \(p_j\), \(w \in \Sigma^*\) is the global watcher which records the sequence of messages that have been transmitted.

For two configurations \(\gamma = (Q_1, t_1, \ldots, Q_n, t_n, w)\), \(\gamma' = (Q_1', t_1', \ldots, Q_n', t_n', w')\), we say that \(\gamma\) derives \(\gamma'\), written as \(\gamma \rightarrow \gamma'\), if one of the following holds:

- A peer \(p_j\) executes an e-move, i.e., there exists \(j \in [1..n]\) s.t. \((t_j, \epsilon, t_j') \in \Delta_j\), and \(Q_j' = Q_j\) and \(w' = w\), and for each \(k \neq j\), \(Q_k' = Q_k\) and \(t_k' = t_k\).
- A peer \(p_j\) consumes an input, i.e., there exist \(j \in [1..n]\) and \(a \in \Sigma_{j}^{\text{in}}\) s.t. \((t_j, a, t_j') \in \Delta_j\), and \(w' = w\), \(Q_j = a Q_j'\), and for each \(k \neq j\), \(Q_k' = Q_k\) and \(t_k' = t_k\).
- A peer \(p_j\) sends an output to peer \(p_k\), i.e., there exist \(j, k \in [1..n]\) and \(b \in \Sigma_{j}^{\text{out}} \cap \Sigma_{k}^{\text{in}}\) s.t. \((t_j, b, t_j') \in \Delta_j\), and \(w' = wb\), \(Q_k' = Q_k b\), and \(Q_j' = Q_j\) for each \(l \neq k\), and \(t_m' = t_m\) for each \(m \neq j\).

For each configuration \(c = (Q_1, t_1, \ldots, Q_n, t_n, w)\), we denote its global watcher content as \(gw(c) = w\). Next we define the key notion of “run” and “conversation”.

Definition 4. Let \(S = (n, \{p_1, \ldots, p_n\}, \sigma)\) be an e-service. A run \(\gamma\) of \(S\) is a finite or infinite sequence of configurations \(\gamma = c_0, c_1, c_2, \ldots\) satisfying the first two of the following conditions, and a complete run is an infinite configuration sequence satisfying all of them.

1. \(c_0 = (\epsilon, s_1, \ldots, s_n, \epsilon)\) (\(s_i\) is the initial state of \(p_i\) for each \(i \in [1..n]\)),
2. For each \(0 \leq i < |\gamma|\), \(c_i \rightarrow c_{i+1}\),
3. For each \(\ell \in [1..n]\) and each \(i \geq 0\), there exist \(j > i, k > i\) such that
   - \(a) t_{\ell}^i\) is a final state, where \(t_{\ell}^i\) is the state of \(p_\ell\) in \(c_j\).
   - \(b) head(Q_{\ell}^j) \neq head(Q_{\ell}^i)\) if \(Q_{\ell}^j \neq \epsilon\), where \(Q_{\ell}^j\) and \(Q_{\ell}^i\) are the queue contents of \(p_\ell\) in \(c_j\) and \(c_k\) respectively.
4. For each \(i \geq 0\), there exists a \(j > i\) such that \(gw(c_i) \neq gw(c_j)\).
An infinite word \( w \in \Sigma^\omega \) is a conversation of \( S \) if there exists a complete run \( c_0, c_1, c_2, \ldots \) of \( S \) such that for each \( i \geq 0 \), \( gw(c_i) \) is a finite prefix of \( w \). Let \( \mathcal{C}(S) \) denote the set of conversations of \( S \).

In Definition 4 condition 3 requires that during a complete run, the Büchi acceptance condition of each peer should be met, and all messages ever sent should be eventually consumed; condition 4 specifies that global message exchange should eventually advance. The notion of conversation captures the global behaviors where each peer proceeds correctly according to its specification. Note that there might be bad runs where some peer is blocked by an unexpected message, or all peers stay in a waiting (deadlock) state.

3 LTL Model Checking

Given an e-service specification, one interesting problem is to check if its conversations satisfy an LTL property. We will first define LTL properties\[12, 11\] on conversations. Then the decidability of LTL model checking will be discussed.

For a conversation \( w = w_0, w_1, w_2, \ldots \), let \( w_i \) denote the \( i \)-th message in \( w \), and \( w^i = w_i, w_{i+1}, w_{i+2}, \ldots \) the \( i \)-th suffix of \( w \). The set of atomic propositions (\( \mathcal{AP} \)) is the power set of messages, i.e., \( \mathcal{AP} = 2^\Sigma \). The syntax and semantics of LTL formulas are defined as follows, where \( \psi \in \mathcal{AP} \) is an atomic proposition, and \( \phi \) and \( \varphi \) are two LTL formulas.

\[
\begin{align*}
  w \models \psi & \iff w_0 \in \psi \\
  w \models \neg \phi & \iff w \not\models \phi \\
  w \models \phi \land \varphi & \iff w \models \phi \text{ and } w \models \varphi \\
  w \models \phi \lor \varphi & \iff w \models \phi \text{ or } w \models \varphi \\
  w \models X\phi & \iff w_1 \models \phi \\
  w \models G\phi & \iff \text{for all } i \geq 0, w^i \models \phi \\
  w \models \phi U \varphi & \iff \text{there exists } j \geq 0, \text{ s.t. } w^j \models \varphi \text{ and, for all } 0 \leq i < j, w^i \models \phi
\end{align*}
\]

We say that an e-service \( S \) satisfies an LTL formula \( \phi \) (denoted as \( S \models \phi \)) if for each conversation \( w \in \mathcal{C}(S) \), \( w \models \phi \).

One natural question concerning model checking an e-service \( S \) is whether we can find a Büchi automaton to characterize the conversation set \( \mathcal{C}(S) \). A positive answer would imply that many verification techniques become immediately available. Unfortunately we show that the answer is negative. Consider the system shown in Fig. 2. In each round of message exchange,
Online Stock Broker sends a list of “Rawdata” to Research Department for further analysis, where for each “Rawdata” one “Data” is generated and sent to Investor. Message classes “EndofRdata”, “Start”, and “Complete” are intended to synchronize the three peers. Finally Investor acknowledges Online Stock Broker with “Ack” so that a new round of data processing can start. This seemingly simple example produces a non \( \omega \)-regular set of conversations. Consider its intersection with an \( \omega \)-regular language \( (R^*_{ESD}CA)^\omega \) (each message is represented by its first letter). It is easy to infer that the result is \( (R^*_{ESD}CA)^\omega \), because Investor enforces that “Start” should arrive earlier than “Data”. By an argument similar to pumping lemma, we can show that this intersection cannot be recognized by any Büchi automata.

**Proposition 5.** There exists an e-service \( S \) such that \( C(S) \) is not accepted by any Büchi automaton.

In fact, given a set of finite state peers LTL model checking is undecidable. The proof can be shown by reduction from the halting problem of Turing Machines. For each Turing Machine \( M \) we can construct a two-peer e-service \( S \) that simulates \( M \) and exchanges a special message (say \( m_t \)) once \( M \) terminates. Thus \( M \) terminates if and only if \( S \models \Sigma U m_t \).

**Theorem 6.** Given an e-service \( S = (n, P, \sigma) \) and an LTL property \( \phi \), determining if \( S \models \phi \) is undecidable.

### 4 Conversation Protocols

By Proposition 5 conversations of an arbitrary e-service are not always \( \omega \)-regular, thus it is natural to consider a “top-down” approach to e-service design by specifying permitted conversations to restrict the global behavior of an e-service. In this section, we introduce the notion of a *conversation protocol* to constrain the global behavior by a nonredundant Büchi automaton. Then we study the concept of realizability: given a Büchi conversation protocol, is it possible to obtain peers to form an e-service which produces exactly the same set of conversations as specified by the protocol? We present three realizability conditions that guarantee a realizable conversation protocol.

To facilitate the technical discussions below, a *peer prototype* is defined as a pair of disjoint sets \((\Sigma_{in}, \Sigma_{out})\). A peer \( p \) implements a peer prototype \((\Sigma_{in}, \Sigma_{out})\) if the input and output alphabets in \( p \) are \( \Sigma_{in}, \Sigma_{out} \) (resp.). Similarly, we can define an e-service prototype as an e-service with peers replaced by peer prototypes, and an e-service \( S \) implements an e-service prototype \( S^P \) if peers in \( S \) implement corresponding peer prototypes in \( S^P \). A standard Büchi automaton \( A \) is *nonredundant* if for every state \( s \) in \( A \) there is a run of some accepted word traveling through \( s \).

**Definition 7.** Let \( S^P = (n, P^P, \sigma) \) be an e-service prototype and \( \Sigma \) be the union of all alphabets in \( S^P \). A conversation protocol over \( S^P \) is a nonredundant Büchi automaton \( A = (\Sigma, T, s, F, \Delta) \) with the alphabet \( \Sigma \). A conversation protocol \( A \) is realizable if there exists an e-service \( S \) which implements \( S^P \) and \( C(S) = L(A) \).
Our definition of realizability here is similar to the weak realizability in [4], where deadlock is not considered. A conversation protocol \( \mathcal{A} \) satisfies an LTL property \( \psi \), written as \( \mathcal{A} \models \psi \) if for all \( w \in \mathcal{L}(\mathcal{A}) \), \( w \models \psi \). The following theorem follows from the well-know results in LTL model checking [12, 11].

**Theorem 8.** Given a conversation protocol \( \mathcal{A} \) for an e-service prototype \( S^P = (n, P^P, \sigma) \) and an LTL property \( \phi \), determining if \( \mathcal{A} \models \phi \) is decidable.

Note that Theorem 8 does not solve our problem, because not every Büchi conversation protocol is realizable. Consider an e-service prototype with four peers, \( p_a, p_b, p_c, p_d \), where \( \Sigma_{\text{out}}^a = \Sigma_{\text{in}}^b = \{ \alpha \} \), \( \Sigma_{\text{out}}^c = \Sigma_{\text{in}}^d = \{ \beta \} \), and \( \Sigma_{\text{out}}^b = \Sigma_{\text{in}}^c = \Sigma_{\text{out}}^d = \emptyset \). The conversation protocol \( \mathcal{A} = (\Sigma, T, s, F, \Delta) \) with \( \Sigma = \{ \alpha, \beta \} \), \( T = \{ 0, 1, 2 \} \), \( s = 0 \), \( F = \{ 2 \} \), and \( \Delta = \{ (0, \alpha, 1), (1, \beta, 2), (2, \beta, 2) \} \) is not realizable, because there is no communication between peers \( p_a \) and \( p_c \), so there is no way for them to make sure that \( \alpha \) is sent before any \( \beta \) is sent.

### 4.1 Realizability Conditions

In the following, we present three realizability conditions that guarantee a realizable conversation protocol. We write \( w_1 \preceq w_2 \) to denote a word \( w_1 \) being a prefix of \( w_2 \) (\( w_1 \) may be equal to \( w_2 \)). Let \( \mathcal{L}_s^*(\mathcal{A}) \) includes all finite prefix of \( \mathcal{L}(\mathcal{A}) \) for a Büchi automaton \( \mathcal{A} \), clearly \( \mathcal{L}_s^*(\mathcal{A}) \) is regular. Given a conversation protocol \( \mathcal{A} \), and a peer (prototype) \( p_i \), the projection of \( \mathcal{A} \) onto \( p_i \) is a Büchi automaton \( \mathcal{A}_i \) obtained from \( \mathcal{A} \) by replacing each move for a message not in the alphabet of \( p_i \) by an \( \epsilon \)-move. We define \( S^P(\mathcal{A}) \) to be the e-service derived from the conversation protocol \( \mathcal{A} \) based on the e-service prototype \( S^P \) where each peer in \( S^P(\mathcal{A}) \) is \( \mathcal{A}_i \). Clearly \( \mathcal{L}(\mathcal{A}_i) = \pi_i(\mathcal{L}(\mathcal{A})) \), where \( \pi_i \) is the projection operator w.r.t. peer \( p_i \).

**Lossless join property** We now introduce the “lossless join” property. Intuitively, the lossless join property requires that a protocol is complete w.r.t. the product of its projection to each peer. Let \( S^P = (n, \{ p_1, \ldots, p_n \}, \delta) \) be an e-service prototype, the JOIN operator is defined as: \( \text{JOIN}(L_1, \ldots, L_n) = \{ w \mid w \in \Sigma^\omega, \forall i \in [1, n], \pi_i(w) \in L_i \} \).

**Definition 9.** Let \( \mathcal{A} \) be a conversation protocol for an e-service prototype \( S^P = (n, P^P, \delta) \). \( \mathcal{A} \) is lossless join if \( \mathcal{L}(\mathcal{A}) = \text{JOIN}(\pi_1(\mathcal{L}(\mathcal{A})), \ldots, \pi_n(\mathcal{L}(\mathcal{A}))) \).

The check of lossless join property is straightforward. Obtain \( S^P(\mathcal{A}) \) from \( \mathcal{A} \), and then construct the Cartesian product (a generalized Büchi automaton) of \( \mathcal{A}_1, \ldots, \mathcal{A}_n \). Then verify whether the resulting product is equivalent to \( \mathcal{A} \).

**Synchronous compatible property** Before we introduce the property, let us revisit the Fresh Market Update example in Fig. 2. We have argued in §3 that the composition in Fig. 2 is bad because the conversation set is not \( \omega \)-regular, which inhibits the application of model checking techniques. In fact it is even worse. Consider the peer Investor, it is possible that “Data” arrives earlier than “Start”,
and Investor is blocked. This scenario is similar to the “illegal states” described in [3], where a peer receives an unexpected message. We define an synchronous compatible property to avoid such scenarios.

**Definition 10.** Let \( \mathcal{A} \) be a conversation protocol for an e-service prototype \( S^P = (n, P, \delta) \). \( \mathcal{A} \) is said to be synchronous compatible if for each word \( w \in \Sigma^* \) and each message \( \alpha \in \Sigma^a \cap \Sigma^b \), the following holds:

\[
(\forall i \in [1,n] \in \pi_i(w) \in \pi_i(L^*_{\leq}(\mathcal{A})) \land \pi_a(w\alpha) \in \pi_a(L^*_{\leq}(\mathcal{A})) \Rightarrow \pi_b(w\alpha) \in \pi_b(L^*_{\leq}(\mathcal{A})))
\]

The decision procedure of synchronous compatible property proceeds as follows: construct e-service \( S^P(\mathcal{A}) \) from \( \mathcal{A} \). For every peer make each state a final state, and then determinize each peer. Construct the Cartesian product of all peers, and check if there is any illegal state, i.e., there is a peer ready to send a message while the receiver is not ready to receive. \( \mathcal{A} \) is not synchronous compatible if an illegal state is found.

**Autonomous property** Synchronous compatible property is still not enough to constrain a conversation protocol. Consider the conversation protocol \( \mathcal{A}_0 \) shown in Fig. 3. It is easy to infer that \( \mathcal{A}_0 \) is synchronous compatible, however the word \( \beta\alpha\gamma\alpha^2 \) is a legal conversation that is not contained in \( L(\mathcal{A}_0) \).

Taking a close look at the execution paths of all peers (denoted by dotted arrows in Fig. 3), we learn that the abnormal conversation is the result of “ambiguous” understanding of the protocol by different peers, and the racing between \( A \) and \( B \) at the initial state is the main cause. Consequently, we introduce the following autonomous property to restrict racing conditions, so that at any point each peer can make independent decisions to receive, to send or to terminate.

Let \( \mathcal{A} \) be a conversation protocol on an e-service prototype \( S^P = (n, P^P, \delta) \). For a peer \( p_i \in P^P \), we say \( p_i \) is output-ready (input-ready) at a word \( w \in \Sigma_i^* \) if there exists a word \( w'\alpha \in L^*_{\geq}(\mathcal{A}) \) such that \( \alpha \) is an output (resp. input) message of \( p_i \) and \( \pi_i(w) = w \). Similarly \( p_i \) is terminate-ready at a word \( w \in \Sigma_i^* \) if there exists a word \( w' \in L(\mathcal{A}) \) such that \( \pi_i(w') = w \).

**Definition 11.** Let \( \mathcal{A} \) be a conversation protocol on e-service prototype \( S^P = (n, P^P, \delta) \). \( \mathcal{A} \) is autonomous if for each peer prototype \( p_i \in P^P \) and for each finite prefix \( w \in L^*_{\geq}(\mathcal{A}) \), \( p_i \) at \( \pi_i(w) \) is only one of the following: output-ready, input-ready, or terminate-ready.
Given a conversation protocol \( \mathcal{A} \) and its e-service prototype \( S^P = (n, P^P, \delta) \), we can check the autonomous property as follows. For each peer \( p_i \), let \( \mathcal{A}_i = (\Sigma^\text{in}_i, \Sigma^\text{out}_i, T_i, s_i, F_i) \) be its peer implementation in \( S^P(\mathcal{A}) \), and let \( T'_i \subseteq T_i \) includes each state \( s \) where an \( \epsilon \) loop starting at \( s \) passes through at least one final state. Construct prefix automaton \( \mathcal{A}^*_i \) for each \( \mathcal{A}_i \) by making each state in \( \mathcal{A}_i \) a final state. Determinize \( \mathcal{A}^*_i \), and now each state of \( \mathcal{A}^*_i \) can be represented by a subset of \( T_i \). We check each state \( s' \) of \( \mathcal{A}^*_i \). When \( s' \cap T'_i \) is not empty, we require that there is no outgoing transitions starting from \( s' \). If \( s' \cap T'_i \) is empty, then the outgoing transitions from \( s' \) are required to be either all output messages or all input messages. The complexity of the above check is \( \text{NPTIME} \) because of the determinization procedure. The following lemma summarizes the complexity of checking three realizability conditions.

**Lemma 12.** Given a conversation protocol \( \mathcal{A} \) and an e-service prototype \( S^P \), it can be determined in polynomial time in the size of \( \mathcal{A} \) and \( S^P \) if \( \mathcal{A} \) has lossless join, synchronous compatible, and autonomous property.

We now proceed to present the main result (Theorem 14), which shows that if the realizability conditions are satisfied, a conversation protocol is realizable.

**Lemma 13.** Let \( \mathcal{A} \) be a synchronous compatible and autonomous conversation protocol, and \( S^P(\mathcal{A}) \) the e-service obtained from the projection of \( \mathcal{A} \) to each peer, then for each conversation \( w \in \text{C}(S^P(\mathcal{A})) \), the following two statements hold

1. for each peer \( p_i \), \( \pi_i(w) \in \pi_i(\mathcal{L}(\mathcal{A})) \), and
2. during any complete run of \( w \), each message sent is consumed eagerly by its receiver, i.e., a peer never sends a message with its queue not empty.

**Proof.** (Sketch) We need only prove statement (2) because (1) is implied by (2). Assume \( \mathcal{A} \) is synchronous compatible and autonomous, but there is a a run \( R \) where a peer \( p_x \) sends out a message \( \alpha_n \) while a message \( \alpha_m \) is stored in its queue. Without loss of generality, let \( \alpha_m \) be the first such message during \( R \). Hence each \( \alpha_i \) where \( i < m \) is consumed eagerly by its receiver. It is not hard to show that for each peer \( p_i \), \( \pi_i(\alpha_0 \ldots \alpha_{m-1}) \in \mathcal{L}(\mathcal{A}_i) \), now by synchronous compatible definition, \( \pi_x(\alpha_1 \ldots \alpha_m) \in \mathcal{L}(\mathcal{A}_x) \), and hence \( p_x \) is input ready at \( \pi_x(\alpha_1 \ldots \alpha_m-1) \). On the other hand, we can infer from the run \( R \) that \( p_x \) is output ready at \( \pi_x(\alpha_1 \ldots \alpha_{m-1}) \) because of message \( \alpha_n \). This contradicts with the autonomous property, and thus the assumption does not hold.

**Theorem 14.** A conversation protocol \( \mathcal{A} \) for an e-service prototype \( S^P \) is realizable, i.e., \( \text{C}(S^P(\mathcal{A})) = \mathcal{L}(\mathcal{A}) \), if \( \mathcal{A} \) is lossless join, synchronous compatible, and autonomous.

Following Lemma 12 and Theorem 14, we get the following verification strategy: (1) A conversation protocol is specified by a realizable Büchi automaton. (2) The properties of the protocol are verified on the Büchi automaton specification, (3) The peer implementations for the conversation protocol are synthesized from the Büchi automaton via projection.
The three realizability conditions in Theorem 14 may seem restrictive, however they are satisfied by many real life e-service applications. We verified that five out of the six examples listed on IBM Conversation Support site [19] satisfy the conditions. In fact, except restricting the racing between send and receive actions, our realizability conditions still allow a certain level of parallelism, which makes it acceptable to many e-services.

The results in this section can be directly applied to the framework in [9], and even better results can be achieved. For an FSA conversation protocol $\mathcal{A}$ and its e-service prototype $S^P$, we can determinize each peer implementation in $S^P(\mathcal{A})$, and it is guaranteed that their composition is deadlock free and peers do not get blocked by unexpected messages in the composition. The difference between the two frameworks is that nondeterministic Büchi automata cannot be determined.

5 Conclusions

In this paper we studied the global behavior of reactive e-services in terms of the conversations permitted by e-services. LTL model checking on e-services specified using the bottom-up approach was shown to be undecidable. This suggests that specifying the permitted conversations as conversation protocols in a top-down fashion is beneficial. However, not every conversation protocol defined by a Büchi automaton is realizable by asynchronous peers. We gave three conditions on conversation protocols which ensure realizability. Studying the global behavior of e-services is a promising research topic. The results in [9] and in this paper provide a starting point.

Acknowledgments

Bultan was supported in part by NSF grant CCR-9970976 and NSF Career award CCR-9984822; Fu was partially supported by NSF grant IIS-0101134 and NSF Career award CCR-9984822; Su was also supported in part by NSF grants IIS-0101134 and IIS-9817432.

References


Boolean Operations
for Attribute-Element Constraints
(Extended Abstract)

Haruo Hosoya¹ and Makoto Murata²

¹ Research Institute for Mathematical Sciences, Kyoto University
hahosoya@kurims.kyoto-u.ac.jp
² IBM Tokyo Research Laboratory
mmurata@trl.ibm.com

Abstract. The history of schema languages for XML is an increase of expressiveness. While early schema languages mainly focused on the element structure, Clark first paid an equal attention to attributes by allowing both element and attribute constraints in a single regular expression. In this paper, we investigate an algorithmic aspect of Clark’s mechanism (called “attribute-element constraints”), namely, intersection and difference operations, which have been proved to be important in static typechecking for XML processing programs. The contributions here are (1) proofs of closure under intersection and difference and (2) algorithm formulations incorporating a “divide-and-conquer” strategy for avoiding an exponential blow-up for typical inputs.

1 Introduction

XML [3] is a standard document format that comes with two notions: data and schemas. Data are tree structures and basically have two major constituents, elements, which are tree nodes forming the “skeleton” of documents, and attributes, which are auxiliary name-value pairs associated with each element. Schemas are a mechanism for imposing constraints on these two structures, and what kinds of constraints they can use is defined by a schema language.

Numerous schema languages have been proposed and their history has been basically how to increase expressiveness, allowing finer- and finer-grained controls to the structure of documents. At first, the main interest of the designers of schema languages (DTD [3], W3C XML Schema [9], and RELAX [23]) was elements, and by now this pursue has mostly converged—most schema languages use regular expressions for describing this part of data. On the other hand, the treatments of attributes have been rather simplistic. For example, DTD allows us to specify each individual attribute to be either required or optional. However, there has been a big demand for a higher expressiveness, in particular, involving interdependency among elements and attributes, e.g., “a person element must have either a name subelement or a name attribute, not both.” Such a constraint either cannot be expressed or requires an exponentially large description in early schema languages.

© Springer-Verlag Berlin Heidelberg 2003
Recently, James Clark, in his schema language TREX [6], proposed a description mechanism called attribute-element constraints for overcoming this shortcoming. The kernel of his proposal is to allow mixture of constraints on elements and those on attributes in a single “regular” expression, thus achieving a uniform and symmetric treatment of two kinds of structure. The expressive power yielded by this mechanism is quite substantial—in particular, the above-mentioned attribute-element interdependency can be represented in a simple and straightforward manner.

In this paper, we pay attention to an algorithmic aspect of attribute-element constraints. Specifically, we investigate intersection and difference operations (i.e., compute a new schema representing the intersection of or difference between given two schemas). Studying these has not only been a tradition in formal language theory, but also is strongly motivated by an important application: static typechecking for XML processing—program analysis technology for detecting all run-time type errors at compile time. Indeed, the recent series of papers have used these operations in crucial parts of their typechecking algorithms [16, 10, 24, 20, 22]. For example, intersection is used in almost all of these languages as the core of type inference mechanisms for their pattern matching facilities; difference is used in XDuce type inference for precisely treating data flow through prioritized pattern clauses [17]; and, perhaps most importantly, inclusion test (which can be done by taking difference and then testing emptiness) is used for “subtyping” in many of the above-mentioned languages.

This work has been carried out in collaboration with committee members of the new schema language RELAX NG [8]. As a result, aiming for closure under booleans, RELAX NG adopted the syntactic restrictions described in this paper.

In this abstract, for the space limitation, we mainly present the intersection algorithm. For the difference, we briefly explain the syntactic restrictions that we need to ensure closure under difference. In addition to these two operations, we have also developed an algorithm for testing inclusion where we do not require the above-mentioned restrictions. All technical details omitted here (including algorithm formulations and correctness proofs) can be found in the full version of this paper [15].

2 Attribute-Element Constraints

In our framework, data are XML documents with the restriction that only elements and attributes can appear. That is, we consider trees where each node is given a name and, in addition, associated with a set of name-string pairs. In XML jargon, a node is called element and a name-string pair is called attribute. For example, the following is an element with name article that has two attributes key and year and contains four child elements—two with authors, one with title, and one with publisher.
The ordering among sibling elements is significant, whereas that among attributes is not. The same name of elements can occur multiple times in the same sequence, whereas this is disallowed for attributes.

Attribute-element constraints describe a pair of an element sequence and an attribute set. Let us illustrate the constraint mechanism. Element expressions describe constraints on elements and are regular expressions on names. For example, we can write

\[ \text{author}^+ \text{title} \text{publisher}? \]

to represent that a permitted sequence of child elements are one or more \text{author} elements followed by a mandatory \text{title} element and an optional \text{publisher} element. (Note that the explanation here is informal: for brevity, we show constraints only on names of attributes and elements. The actual constraint mechanism formalized later can also describe contents of attributes and elements.)

Attribute expressions are constraints on attributes and have a notation similar to regular expressions. For example,

\[ @\text{key} @\text{year}? \]

requires a \text{key} attribute and optionally allows a \text{year} attribute. (We prepend an at-sign to each attribute name in order to distinguish attribute names from element names.)

Attribute expressions are different from usual regular expressions in three ways. First, attribute expressions describe (unordered) sets and therefore concatenation is commutative. Second, since names cannot be duplicated in the same set, we require expressions to permit only data that conform to this restriction (e.g., \((@a | @b)@a?\) is forbidden). Third, for the same reason, repetition (\(^+\)) is disallowed in attribute expressions. We provide, however, “wild-card” expressions that allow an arbitrary number of arbitrary attributes from a given set of names (discussed later).

Attribute-element expressions or compound expressions allow one expression to mix both attribute expressions and element expressions. For example, we can write

\[ @\text{key} @\text{year}? \text{author}^+ \text{title} \text{publisher}? \]

to require both that the attributes satisfy \( @\text{key} @\text{year}? \) and that the elements satisfy \( \text{author}^+ \text{title} \text{publisher}? \). The next example is a compound expression allowing either a \text{key} attribute or a \text{key} element, not both.

\[ (@\text{key}|\text{key})@\text{year}? \text{author}^+ \text{title} \text{publisher}? \]
In this way, we can express constraints where some attributes are interdependent with some elements. (Note that we can place attribute expressions anywhere—even after element expressions.)

In addition to the above, we provide “multi-attribute expressions,” which allow an arbitrary number of attributes with arbitrary names chosen from a given set of names. Multi-attribute expressions are useful in making a schema “open” so that users can put their own pieces of information in unused attributes. For example, when we want to require \texttt{key} and \texttt{year} attributes but optionally permit any number of any other attributes, we can write the following expression (where \texttt{(*\key\year)} represents the set of all names except \texttt{key} and \texttt{year}).

\[
\@\texttt{key} \@\texttt{year} \@(*\texttt{key}\texttt{year})^\ast
\]

Although our formulation will not include a direct treatment, multi-attribute expressions can be even more useful if combined with name spaces. (Name spaces are a prefixing mechanism for names in XML documents; see [2] for the details.) For example, when we are designing a schema in the name space \texttt{myns}, we can write the following to permit any attributes in different name spaces (where \texttt{myns:}\ast means “any names in name space \texttt{myns}”).

\[
\@\texttt{myns:}\texttt{key} \@\texttt{myns:}\texttt{year} \@(*\texttt{(myns:}\ast))^\ast
\]

Apart from the kinds of name sets described above, the following can be useful:
(1) the set of all names, (2) the set of all names in a specific name space, and (3) the set of all names except those from some specific name spaces. (In fact, these are exactly the ones supported by RELAX NG [8].)

\section{Data Model}

We assume a countably infinite set \(\mathcal{N}\) of \textit{names}, ranged over by \(a, b, \ldots\). We define \textit{values} inductively as follows: a \textit{value} \(v\) is a pair \((\alpha, \beta)\) where

\begin{itemize}
  \item \(\alpha\) is a set of pairs of a name and a value, and
  \item \(\beta\) is a sequence of pairs of a name and a value.
\end{itemize}

A pair in \(\alpha\) and a pair in \(\beta\) are called \textit{attribute} and \textit{element}, respectively. In the formalization, attributes associate names with values and therefore may contain elements. This may appear odd since XML allows only strings to be contained in attributes. Our treatment is just for avoiding the need to introduce another syntactic category for attribute contents and thereby simplifying the formalism.

We write \(\epsilon\) for an empty sequence and \(\beta_1\beta_2\) for the concatenation of sequences \(\beta_1\) and \(\beta_2\). For convenience, we define several notations for values.

\[
\begin{align*}
  a[v] & \equiv (\emptyset, \langle a, v \rangle) \\
  (\alpha_1, \beta_1)(\alpha_2, \beta_2) & \equiv (\alpha_1 \cup \alpha_2, \beta_1\beta_2) \\
  \emptyset[v] & \equiv (\emptyset, \langle a, v \rangle) \\
  \epsilon & \equiv (\emptyset, \epsilon)
\end{align*}
\]

For example, \(\@a[v] \@b[w] c[u]\) means \(\langle \langle a, v \rangle, \langle b, w \rangle, \langle c, u \rangle \rangle\). We write \(V\) for the set of all values.
2.2 Expressions

Let $S$ be a set of sets of names where $S$ is closed under boolean operations. In addition, we assume that $S$ contains at least the set $N$ of all names and the empty set $\emptyset$. Each member $N$ of $S$ is called name set.

We next define the syntax of expressions for attribute-element constraints. As already mentioned, our expressions describe not only top-level names of elements and attributes but also their contents. Since, moreover, we want expressions to describe arbitrary depths of trees, we introduce recursive definitions of expressions, that is, grammars.

We assume a countably infinite set of variables, ranged over by $x, y, z$. We use $X, Y, Z$ for sets of variables. A grammar $G$ on $X$ is a finite mapping from $X$ to compound expressions. Compound expressions $c$ are defined by the following syntax in conjunction with element expressions $e$.

$$
e ::= @N[x]^+ \quad e ::= N[x]$$

$$c ::= c \mid c \quad e ::= e \mid e$$

$$c ::= c \cdot c \quad e ::= e \cdot e$$

We call the form $@ N[x]^+$ multi-attribute expression (as mentioned) and $@ N[x]$ single-attribute expression. We define $\text{FV}(c)$ as the set of variables appearing in $c$ and $\text{FV}(G)$ as $\bigcup_{x \in \text{dom}(G)} F(x)$. We require any grammar to be “self-contained,” i.e., $\text{FV}(G) \subseteq \text{dom}(G)$, where $\text{dom}(G)$ is the domain of $G$. In the sequel, we use the following shorthands.

$$@a[x]^+ \equiv @\{a\}[x]^+ \quad @a[x] \equiv @\{a\}[x]$$

$$a[x] \equiv \{a\}[x] \quad c^+ \equiv c^+ | \epsilon$$

$$c? \equiv c | \epsilon \quad @N[x]^* \equiv @N[x]^+ | \epsilon$$

We forbid concatenation of expressions with overlapping attribute name sets. That is, we first define $\text{att}(c)$ as the union of all the attribute name sets (the $N$ in the form $@N[x]^+$ or $@N[x]$) appearing in the expression $c$. Then, any expression must not contain an expression $c_1 c_2$ with $\text{att}(c_1) \cap \text{att}(c_2) \neq \emptyset$. We define $\text{elm}(c)$ as the union of all the element name sets (the $N$ in the form $N[x]$) appearing in the expression $c$.

The semantics of expressions with respect to a grammar is described by the relation of the form $G \vdash v \in c$, which is read “value $v$ conforms to expression $c$ under $G$.” This relation is inductively defined by the rules in Figure 1. Note that rules T-Alt and T-Cat treat alternation and concatenation both in compound expressions and element expressions.

3 Boolean Algorithms

In this section, we present our algorithms for intersection and (briefly) difference for attribute-element grammars.
∀i. (αi ∈ N ⋄ G ⊢ v1 ∈ G(x))  k ≥ 1  αi ≠ αj for i ≠ j

∀i. G ⊢ a1[v1] ... ak[vk] ∈ @N[x] T-ATTREP

G ⊢ v ∈ G(x)  a ∈ N  a ∈ @a[a] ∈ @N[x] T-ATT

G ⊢ v ∈ G(x)  a ∈ N  a[v] ∈ N[x] T-ELM

G ⊢ v ∈ c1  or  G ⊢ v ∈ c2  G ⊢ v ∈ c1 | c2 T-ALT

G ⊢ v1 ∈ c1  G ⊢ v2 ∈ c2  G ⊢ v1 ∈ c1 | c2 T-CAT

G ⊢ v ∈ c  k ≥ 1  ∀i. G ⊢ vi ∈ c  G ⊢ v1 ... vk ∈ c T-PLU

Fig. 1. Semantics

3.1 Partitioning

The key technique in our algorithms is partitioning. Consider first the following intersection of compound expressions.

( @a[x] | a[x] ) ( @b[x] | b[x] ) ∩ @a[y] ( @b[y] | b[y] )

How can we calculate this intersection? A naive algorithm would separate constraints on attribute sets and those on element sequences

( @a[x] @b[x] | @a[x] b[x] | a[x] @b[x] | a[x] b[x] )

and compute the intersection of every pair of clauses on both sides. Such use of “distributive laws” makes the algorithm easily blow up. Fortunately, we can avoid it in typical cases. Note that each expression in the formula (1) is the concatenation of two subexpressions, where the left subexpressions on both sides contain the names @a and a and the right subexpressions contain the different names @b and b. In such a case, we can compute intersections of the left subexpressions and of the right subexpressions separately, and concatenate the results:

( ( @a[x] | a[x] ) ∩ @a[y] ) ( ( @b[x] | b[x] ) ∩ ( @b[y] | b[y] ) )

The intuition behind why this works is that each “partitioned” expression can be regarded as cross products, and therefore the intersection of the whole expressions can be done by intersecting each corresponding pair of subexpressions. Note also that no subexpression is duplicated by this partitioning process. Therefore the algorithm proceeds linearly in the size of the inputs as long as partitioning can be applied. This idea of splitting expressions into orthogonal parts was inspired by Vouillon’s unpublished work on shuffle expressions [25]. We will discuss the difference of our work from his in Section 4.
For treating partitioning in our formalization, it is convenient to view a nested concatenation of expressions as a flat concatenation and ignore empty sequences (e.g., view (c_1 (c_2 c_3) c_4 as c_1 c_2 c_3). In addition, we would like to treat expressions to be “partially commutative,” that is, concatenated c_1 and c_2 can be exchanged if one of them is element-free. For example, the expression \(a(x) \cap b(x) \cap [b(x)]\) is equal to \((a(x) \cap b(x)] \cap [b(x)]\). On the other hand, \((a(x) \cap [a(x)] \cap [b(x)]\) is not equal to \((a(x) \cap [b(x)] \cap [a(x)]\) since, this time, \(a(x)\) prevents such an exchange.

Formally, we identify expressions up to the relation \(\equiv\) defined as follows: \(\equiv\) is the smallest congruence relation including the following.

\[
c_1 c_2 \equiv c_2 c_1 \quad \text{if } \text{elm}(c_1) = \emptyset
\]

\[
c_1 c_2 c_3 \equiv (c_1 c_2) c_3
\]

\[
c \epsilon \equiv c
\]

Now, \((c'_1, c''_1), \ldots, (c'_k, c''_k)\) is a partition of \(c_1, \ldots, c_k\) if

\[
c_i = c'_i c''_i \quad \text{for all } i
\]

\[
(\bigcup_i \text{att}(c'_i)) \cap (\bigcup_i \text{att}(c''_i)) = \emptyset
\]

\[
(\bigcup_i \text{elm}(c'_i)) \cap (\bigcup_i \text{elm}(c''_i)) = \emptyset.
\]

That is, each \(c_i\) can be split into two subexpressions such that the names contained in all the first subexpressions are disjoint with those contained in all the second subexpressions. We will use partition of two expressions \((k = 2)\) in the intersection algorithm and that of an arbitrary number of expressions in the difference. The partition is said proper when \(0 < w(c'_i) < w(c_i)\) for some \(i\). Here, the function \(w\) counts the number of expressions that are concatenated at the top level (except \(c\)). That is, \(w(c) = 0, w(c_1 c_2) = w(c_1) + w(c_2), \) and \(w(c) = 1\) if \(c \neq c\) and \(c \neq c_1 c_2\). (Note that \(\equiv\) preserves \(w\).) This properness will be used for ensuring the boolean algorithms to make a progress.

### 3.2 Intersection

Let grammars \(F\) on \(X\) and \(G\) on \(Y\) be given. We assume that \(F\) and \(G\) have been normalized. That is, the given grammars have already been transformed so that all name sets appearing in them are pair-wise either equal or disjoint. The reason for doing this is to simplify our boolean algorithms. For example, in computing the intersection \(\hat{\cap} N_1[x] \cap N_2[x] \cap N_3[y] \cap N_4[y]\), if \(N_1\) and \(N_2\) are respectively equal to \(N_3\) and \(N_4\), then this intersection is obvious. However, if these are overlapping in a non-trivial way (e.g., \(\hat{\cap} \{a, b\}[x] \cap \{c, d\}[x] \cap \{a, c\}[y] \cap \{b, d\}[y]\)), it would require more work. An actual algorithm for normalization is presented in the full version of this paper [15].

Our intersection algorithm is based on product construction. From \(F\) and \(G\), we compute a new grammar \(H\) on \(X \times Y\) that satisfies

\[
H((x,y)) = \text{inter}(F(x), G(y))
\]
for all $x \in X$ and $y \in Y$. The function $\text{inter}$ computes an intersection of compound expressions. It works roughly in the following way. We proceed the computation by progressively decomposing the given compound expressions. At some point, they become attribute-free. Then, we convert the expressions to element automata (defined later), compute an intersection by using a variant of the standard automata-based algorithm, and convert back the result to an expression. Formally, $\text{inter}$ is defined in Figure 2. The base cases are handled by rules 1 through 6, where each of the arguments is either an element expression (as indicated by the metavariables $e$ or $f$) or a single- or multi-attribute expressions. In rule 1, where both arguments are element expressions, we pass them to another intersection function $\text{inter}^{\text{reg}}$ specialized to element expressions. This function will be explained below. Rules 2 and 3 return $\emptyset$ since the argument expressions obviously denote disjoint sets. Rule 4, 5, and 6 handle the cases where each argument is a single- or multi-attribute expression with the same name set $N$. When both arguments are multi-attributes, rule 4 yields a multi-attribute where the content is the intersection of their contents $x$ and $y$. When either argument is a single-attribute, rule 5 or 6 returns a single-attribute. (Note that, in rules 4 to 6, normalization ensures that the name sets in the given expressions are equivalent.) The inductive cases are handled by rules 7 and 8. Rule 7 applies the partitioning technique already explained. Rule 8 simply expands one union form appearing in the argument expressions.

The intersection function $\text{inter}^{\text{reg}}$ performs the following: (1) construct element automata $M_1$ and $M_2$ from element expressions $e_1$ and $e_2$, (2) compute the “product automaton” $M$ from $M_1$ and $M_2$, and (3) convert $M$ back to an element expression $e$. Element automata are defined as follows. First, an automaton $M$ on an alphabet $\Sigma$ is a tuple $(Q, q^{\text{init}}, Q^{\text{fin}}, \delta)$ where $Q$ is a finite set of states, $q^{\text{init}} \in Q$ is an initial state, $Q^{\text{fin}} \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation [13]. Then, an element automaton is an automaton over $\{N[x] \mid N \in S, x \in X\}$, where $S$ is a set of name sets and $X$ is a set of variables. Since well-known conversion algorithms between automata and regular expressions can directly be used for the case of element automata.
and element expressions by assuming $N[x]$ as symbols, we use them for (1) and (3) parts of the inter$_{reg}$ function.

The product construction for element automata (used for the (2) part of inter$_{reg}$) is slightly different from the standard one. Usually, product construction generates, from two transitions with the same label in the input automata, a new transition with that label in the output automaton. In our case, we generate, from a transition with label $N[⟨x, y⟩]$ and another with label $N[y]$, a new transition with label $N[⟨x, y⟩]$. Formally, given two element automata $M_i = (Q_i, q_i^{init}, Q_i^{fin}, \delta_i)_{N[⟨x_i, y_i⟩]}$ for $i = 1, 2$, the product of $M_1$ and $M_2$ is an automaton $(Q_1 \times Q_2, (q_1^{init}, q_2^{init}), Q_1^{fin} \times Q_2^{fin}, \delta)$ on $\{N[⟨x_1, x_2⟩] | N \in S, x_1 \in X_1, x_2 \in X_2\}$ where

$$\delta = \{((q_1, q_2), N[⟨x_1, x_2⟩], (q'_1, q'_2)) | (q_i, N[x_i], q'_i) \in \delta_i, for i = 1, 2\}.$$

(Note that we use here the assumption that the name sets of elements in given grammars have been normalized.)

We can prove the following expected property for our intersection algorithm.

**Theorem 1.** Let $H(⟨x, y⟩) = \text{inter}(F(x), G(y))$. Then, inter$(c, d) = b$ implies that $H \vdash v \in b$ iff $F \vdash v \in c$ and $G \vdash v \in d$.

The intersection algorithm takes at most a quadratic time in the numbers of variables in the given grammars. However, for each pair of variables, it takes an exponential time in the size of the expressions assigned to the variables in the worst case, where the function needs to fully expand the expressions by using rule 8. There is no other exponential factor in this algorithm. (The function inter$_{reg}$ can be computed in a polynomial time since each of the three steps is polynomial.)

### 3.3 Difference

Our expressions, as they are defined as in Section 2, do not have closure under difference. The kinds of expressions that break the closure are single-attributes $@N[x]$ and multi-attributes $@N[x]^+$ where $N$ is infinite (in both cases). For single-attributes, consider the difference $@N[any]^+ \setminus @N[any]$ (where $N$ is the name set containing all names). This would mean zero, two, or more attributes with any name and any content. However, “two or more” is not expressible in our framework. For multi-attributes, consider the difference $@N[any]^+ \setminus @N[x]^+$ where $N$ is infinite. The resulting expression should satisfy the following. Each value in it has a set of attributes all with names from $N$. But at least one of them has a content not satisfying $x$. The expression $@N[x]^+$ is not a right answer because it requires all attributes to have contents not satisfying $x$.

For this reason, we impose two syntactic restrictions. First, the name set of a single-attribute expression must be a singleton. Note that, with this restriction, we can still represent the case where $N$ is finite\footnote{RELAX NG adopts this restriction of finiteness.}: when $N = \{a_1, \ldots, a_k\}$

$$@N[x] \equiv @a_1[x] | \ldots | @a_k[x]$$
On the other hand, the case that \( N \) is infinite is not expressible. We consider, however, that this restriction is acceptable from practical point of view since an infinite name set is usually used for representing “arbitrary names from a specific name space” and one naturally wants an arbitrary number of attributes from such a name set. The second restriction is that the content of a multi-attribute expression must be a distinguished variable \textbf{any} accepting any values. We assume that any given grammar \( G \) has the following mapping.

\[
G(\text{any}) = N[\text{any}]^* \circ N[\text{any}]^*
\]

We can still represent the case where \( N \) is finite and the content is not \textbf{any}:

when \( N = \{a_1, \ldots, a_k\} \), we rewrite \( \circ N[x]^+ \) by

\[
\bigcup_{\emptyset \neq \{i_1, \ldots, i_p\} \subseteq \{1, \ldots, k\}} @a_{i_1}[x] \ldots @a_{i_p}[x].
\]

On the other hand, we cannot handle the case where \( N \) is infinite and the content is not \textbf{any}. However, this restriction seems reasonable since an open schema typically wants both the name and the content to be generic—making only the content specific seems rather unnatural.

We can easily see that these restrictions do not break closure under intersection (details are omitted).

Our difference algorithm uses the previously mentioned partitioning technique in conjunction with a “top-down” form of subset construction \([18, 17, 11]\). Further details can be found in \([15]\).

4 Related Work

Our study on attribute constraints has a strong relationship to type theories for record values (i.e., finite mappings from labels to values). Early papers presenting type systems for record types do not consider the union operator and therefore no such complication arises as in our case. (A comprehensive survey of classical records can be found in \([12]\).) Buneman and Pierce have investigated record types with the union operator \([5]\). Their system does not, however, have any mechanism similar to our multi-attribute expressions or recursion. Frisch, Castagna, and Benzaken \([11]\) have designed a typed XML processing language CDuce that supports attribute types based on records. Although the descriptive power of their types is the same as ours, type expressions to represent interdependency between attributes and elements are exponentially larger than ours since they do not allow mixture of element and attribute constraints. The DSD schema language \([19]\), designed by Klarlund, Möller, and Schwartzbach, can also express the kinds of attribute-element interdependencies discussed here. However, this schema language is not closed under intersection or complementation. Its descendent DSD2 \([21]\) has the expressiveness of concern. Its closure properties are now under investigation by the designers.

In his unpublished work, Vouillon has considered an algorithm for checking the inclusion relation between shuffle expressions \([25]\). His strategy of progressively decomposing given expressions to two orthogonal parts made much
influence on our boolean and inclusion algorithms. The difference is that his algorithm directly answers yes or no without constructing new grammars like our case, and therefore does not incur the complication of switching back and forth between the expression representation and the automata representation.

Another important algorithmic problem related schemas is validation. There have been several validation algorithms proposed for attribute-element constraints. One is designed and implemented by Clark [7] based on derivatives of regular expressions [4, 1]. Another is presented by Hosoya and Murata using so-called attribute-element automata [14].

5 Future Work

In this paper, we have presented our intersection and difference algorithms. We have already implemented them in the typed XML processing language XDupe [16]. For the examples that we have tried, the performance seems quite reasonable. We plan to collect and analyze data obtained from the experiment on the algorithms in the near future. We also feel that we need some theoretical characterization of the algorithms. In particular, our algorithms contain potentials of blow up in many places. Although our implementation techniques presented in the full version of this paper [15] have been sufficient for our examples, one would like to have some more confidence.

Acknowledgments

We learned a great deal from discussion with James Clark, Kohsuke Kawaguchi, Akihiko Tozawa, Benjamin Pierce, and members of the Programming Languages Club at University of Pennsylvania. We are also grateful to anonymous PLAN-X referees, who help us improving this paper substantially. Hosoya has been partially supported by Japan Society for the Promotion of Science and Kayamori Foundation of Informational Science Advancement while working on this paper.

References

XML Schema Containment Checking Based on Semi-implicit Techniques

Akihiko Tozawa¹ and Masami Hagiya²

¹ IBM Research, Tokyo Research Laboratory, IBM Japan ltd., Japan
atozawa@trl.ibm.com
² Graduate School of Information Science and Technology
University of Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp

Abstract. XML schemas are computer languages defining grammars for XML (Extensible Markup Languages) documents. Containment checking for XML schemas has many applications, and is thus important. Since XML schemas are related to the class of tree regular languages, their containment checking is reduced to the language containment problem for non-deterministic tree automata (NTAs). However, an NTA for a practical XML schema has $10^2 - 10^3$ states for which the textbook algorithm based on naive determinization is expensive. Thus we in this paper consider techniques based on BDDs (binary decision diagrams). We used semi-implicit encoding which encodes a set of subsets of states as a BDD, rather than encoding a set of states by it. The experiment on several real-world XML schemas proves that our containment checker can answer problems that cannot be solved by previously known algorithms.

1 Introduction

This paper discusses the containment checking for XML schemas, which is essentially the problem of the containment checking between two non-deterministic tree automata (NTAs). We use reduced and ordered BDDs (ROBDDs) to solve this problem.

In this paper, we refer to a standard bottom-up automaton on binary trees as NTA. In the problems of our interest, each NTA involves $10^2 - 10^3$ states. Although this is not a large number for word automata, the situation is different with NTA. For instance, the determinization of NTAs, which we may use to compute the complement automaton, is often very expensive. Thus the textbook algorithm for automata containment does not work as it is. On the other hand, our containment algorithm uses semi-implicit techniques. That is, we do not perform determinization explicitly but rather we do so by encoding a set of subsets of the state set of the NTA (= a set of states of the determinized automaton) with a BDD.

In symbolic (or implicit) verification techniques [CGP99], usually a set of states is encoded by a single BDD, that is, we use a bit vector encoding for each state. In contrast, our technique is called semi-implicit, since we do not encode
each state, but rather we encode each subset of the state set, and thus each BDD represents a set of such subsets. Semi-implicit techniques are not used in previous work on the language containment, since they are considered not to be efficient. However, our algorithm is efficient. The reason for this efficiency lies in the use of positive BDDs. A positive BDD represents an upward closures of a set of subsets. It is safe to use positive BDDs because each subset introduced in the upward closure is always a weaker condition with respect to whether or not the containment holds. By restricting BDDs to positive ones, we further reduce the number and size of BDDs appearing in the analysis.

Background

Our interests in NTAs come from their relation to XML schemas, i.e., grammar specification languages for XML (Extensible Markup Language). For example, a grammar for XHTML (whose complete version is defined by W3C [Wor00]) is approximately as follows:

\[
S ::= \text{html}(Head, Body) \\
Head ::= \text{head}(Meta^*, Title, Meta^*) \\
Body ::= \text{body}((H1|H2|\ldots)^*) \\
\ldots
\]

The grammar means that we at top have an html-node, in which, i.e., between \(<html>\) and \(</html>\) tags, we have head and body nodes, ... and so on. The class of languages that grammars as above define is identical to the class of regular tree languages [HM02, HVP00, MLM01]. The software that checks if XML documents comply with a schema is called a validator. A validation corresponds to the execution of a tree automata in correspondence.

The ultimate goal of our study is to develop efficient and innovative tools and softwares for XML and its schemas. Currently only established technologies for XML schemas are validators, but there are other applications. For some applications, it is important to investigate problems with higher complexity. For example, type checkers for XML processing languages [CMS02, HVP00] often use set operations on types which are essentially boolean operations on automata. In particular, they usually define subtyping relation between two types by means of the language containment testing for NTAs, which in the worst case requires \(\text{EXPTIME}\) to the size of states [Sei90].

Outline

In the next section, we overview the related work both from verification technologies and XML technologies. Sections 3 describes preliminary definitions and concepts. In Section 4, we discuss a containment algorithm for NTAs. We also show some experimental results on examples including those from real-world XML schemas. Section 5 discusses the future work.
2 Related Work

Automata and BDDs

There have been several analyses of automata based on BDDs in the context of verification technology. In this context, automata are on infinite objects. Existing symbolic algorithms for \( \omega \)-automata containment either restrict automata to deterministic ones as in Touati, et al. [TBK95] (so that there is a linear-size complement automaton), or use intricate BDD encoding as in Finkbeiner [Fin01] and Tasiran, et al. [THB95]. None of these algorithms use semi-implicit encoding similar to ours. This may be because their automaton usually represents a Cartesian product of concurrent processes, where the number of states grows exponential to the number of processes. The large number of states makes the semi-implicit encoding almost impossible. This problem does not apply to our automata modeling XML schemas.

Mona's approach [HJJ+95] is also based on BDDs. In Mona, transition functions are efficiently expressed by a set of multi-terminal BDDs each corresponding to a source state. The nodes of each BDD represent boolean vector encoding of labels of transitions and its leaves represent the target states. We have not tested whether this encoding is also valid with problems of our interests. They also extended their representation to deal with NTAs [BKR97].

NTA Containment

Hosoya, et al. proposed another NTA containment algorithm [HVP00] which is one of the best algorithms that can be used in a real-world software. Hosoya, et al.'s algorithm is based on the search of proof trees of the co-inductively defined containment relation. The algorithm is explicit, i.e., it does not use BDDs. It proceeds from the final states of bottom-up tree automata to the initial states. We later compare our algorithm to theirs.

Kuper, et al. [KS01] proposed the notion of subsumption for XML schemas. Subsumption, based on the similarity relation between two grammar specifications, is a strictly stronger relation than the containment relation between two languages. We do not deal with subsumption in this paper.

Shuffle and Other Topics of XML Schemas

The word XML schema is not a proper noun. Indeed, we have a variety of XML schema languages including DTD, W3C XML Schema, RELAX NG [Ora01], DSD [KSM02], etc. The result of this paper can directly be applied to schemas that can easily be transformed into NTAs. Such schemas include regular expression types [HVP00], RELAX [rel] and DTDs.

Hosoya, et al. are working on containment of shuffle regular expressions and they have some preliminary unpublished results. Hosoya and Murata [HM02] also proposed a containment algorithm for schemas with attribute-element constraints. We do not discuss shuffle regular expressions and attribute-element
constraints in this paper, but they are also important in XML schema languages such as RELAX NG [Ora01]. We just mention the result of Mayer and Stockmeyer [MS94] stating that the containment of shuffle regular expressions is EXPSPACE-complete. This means the problem of containment for RELAX NG is essentially harder than the problem dealt with in this paper.

3 Preparation

3.1 Binary Trees

As noted in the introduction, we model XML schemas defining a set of XML documents by automata defining a set of binary trees. Indeed, we can view an XML document instance as a binary tree. To clarify the relationship between binary trees and XML trees, we here introduce a special notation for binary trees\(^1\).

Throughout the paper, we use \(u, v, w, \ldots\) to range over a set of binary trees. In this paper, a binary tree \(v\) on the alphabet \(\Sigma\) (we use \(a\) to range over \(\Sigma\)) is represented by a sequence

\[a_0(v_0)a_1(v_1) \cdots a_{n-1}(v_{n-1})\]

of nodes \(a_i(v_i)\), where each \(a_i\) is a label and each \(v_i\) is a subtree. We have no restriction on the number \(n\) of nodes in a sequence. We use \(\epsilon\) to denote a null tree (\(n = 0\)). If \(n > 0\), we split a sequence \(v\) into the first node \(a(u) (= a_0(v_0))\) and the remainder \(w (= a_1(v_1) \cdots a_{n-1}(v_{n-1}))\). Thus a sequence \(v\) is either in the form \(a(u)w\) or \(\epsilon\), i.e., it is a binary tree.

3.2 ROBDD

We use reduced and ordered BDDs (ROBDDs) by Bryant [Bry86]. Each BDD over a variable set \(X\) represents a boolean function over \(X\). A boolean function takes an assignment of a truth value (0 or 1) to each variable, and returns again a truth value. Note that each assignment also represents a subset of \(X\) such that the subset consists of variables to which 1 is assigned. Thus, a boolean function also represents a set of subsets, i.e., an element of \(2^{2^X}\), such that each subset represents an assignment which gives the return value 1.

Let \((X, <)\) be an arbitrary linearly ordered set, and \(x\) be an element of \(X\). A BDD \(\alpha\) is defined as follows:

\[\alpha ::= (x, \alpha, \alpha) \mid 0 \mid 1\]

If \(\alpha\) is a node \((x, \beta, \gamma)\), we can use the notation \(\alpha.\text{var} = x\), \(\alpha.\text{l} = \beta\) and \(\alpha.\text{h} = \gamma\) to obtain the content of \(\alpha\)\(^2\). Otherwise \(\alpha\) is called a leaf, i.e., a 0-leaf or 1-leaf.

In this paper, we interpret the semantics of a BDD as a set of subsets.

---

1. This notation defines what is often called hedges [MLM01].

2. According to the BDD vocabulary, they are the variable, the node on a low edge, and the node on a high edge of \(\alpha\), respectively.
**Definition 1.** A BDD $\alpha$ represents a set $[\alpha]$ of subsets of $X$:

$$
[0] = \emptyset, \\
[1] = 2^X, \\
[(x, \alpha, \beta)] = \{S \mid S \in [\alpha], x \notin S\} \cup \{S \mid S \in [\beta], x \in S\}
$$

For example, $(x, 0, 1)$ denotes a set of $S$ such that $x \in S$ and $(x, 1, 0)$, a set of $S$ such that $x \notin S$. These two correspond to boolean functions $f(x_1, \ldots, x_n) = x_k$ and $f(x_1, \ldots, x_n) = \neg x_k$, respectively, where $X = \{x_1, \ldots, x_k, \ldots, x_n\}$.

We say a BDD is ordered if all variables appear to the given linear order $<$, i.e., for any subnode $(x, \alpha, \beta)$, each variable $y$ appearing in $\alpha$ or $\beta$ satisfies $x < y$. We say a BDD is reduced if there are no subnodes in the form $(x, \alpha, \alpha)$.

AB D D is an ROBDD if it is reduced and ordered. Each ROBDD is canonical, meaning that for each element of $2^{2^X}$, there is one and only one ROBDD that represents it. In this paper, we denote a set of ROBDDs over $X$ by ROBDD($X$).

We denote by the node size of a BDD, the number of syntactically distinct subnodes in the BDD. The node size of a BDD is often substantially smaller than the cardinality of the underlying set that this BDD represents.

### 4 Containment of Non-deterministic Tree Automata

#### 4.1 Non-deterministic Tree Automata

A non-deterministic tree automaton $A$ is a tuple $\langle \Sigma_A, Q_A, I_A, F_A, \delta_A \rangle$, where $\Sigma_A$ is an alphabet, $Q_A$ is a state set, $\delta_A$ is a transition function that maps $\Sigma_A \times Q_A \times Q_A$ to $2^{Q_A}$, $I_A$ is an initial state set and $F_A$ is a final state set. $\Sigma_A$ will not change throughout the paper and is denoted by $\Sigma$.

**Definition 2.** The language $[q]_A$ corresponding to each state $q$ of $A$ is a set of binary trees defined by the following inductive definition:

- For $q \in I_A$, we have $\epsilon \in [q]_A$.
- If $v_1 \in [q_1]_A$, $v_2 \in [q_2]_A$ and $q_0 \in \delta_A(a, q_1, q_2)$, we have $a(v_1)v_2 \in [q_0]_A$.

An NTA $A$ accepts a binary tree $v$, if $v$ is included in $[q]_A$ for a certain final state $q$. The set of binary trees accepted by $A$ is called the language of $A$ and denoted by $[A]$.

$$
[A] = \bigcup_{q \in F_A} [q]_A
$$

---

3 Usually reducedness also requires node-sharing in a graph representing BDD structures. In our definition, we distinguish BDDs up to their syntactical equalities, and thus subnodes are already shared.
NTAs define an identical class of languages as regular expression types by Hosoya et al \[HVP00\]. This class of languages is called \textit{regular}. In implementation, we used Hosoya et al’s algorithm that converts XML schemas into tree automata.

\textbf{Example 1.} An automaton \(A\), such that \(Q_A = \{0,1,2\}\), \(\delta_A(r,1,0) = \{2\}\), \(\delta_A(a,0,1) = \{1\}\), \(I_A = \{0,1\}\) and \(F_A = \{2\}\), accepts any binary tree of the form \(r(a^n)(n \geq 0)\).

\subsection{Highlight}

We propose an algorithm that given two tree automata \(A\) and \(B\), decides if \([A] \subseteq [B]\) holds or not.

In a traditional algorithm, we complement the automaton at the right hand side, and compute the difference automaton for \(A \cap \bar{B}\). In complementation, we determinize \(B\) using subsets of \(Q_B\) as new states. To get the difference, starting from each pair of an initial state of \(A\) and the initial state set of \(B\), we enumerate all reachable state pairs in \(Q_A \times 2^{Q_B}\) according to a transition in \(A\), and a corresponding set of transitions in \(B\) using the same label. At the final step, if each state pair \((q,S)\) having \(q\) in \(F_A\) satisfies \(S \cap F_B \neq \emptyset\), i.e., \(S\) is not a final state of \(\bar{B}\), the automaton \(A \cap \bar{B}\) is empty, and thus the containment holds.

Our algorithm can similarly be applied to word automata. For simplicity, we here use word automata as an example. To check containment between two automata \(A\) and \(B\) for regular expressions \((a|b)c\) and \(ab|ac|bc\),

\[
\begin{array}{c}
\begin{tikzpicture}[scale=0.5]
  \node[shape=circle,draw] (a) at (0,0) {a};
  \node[shape=circle,draw] (b) at (1,0) {b};
  \node[shape=circle,draw] (c) at (1,-1) {c};
  \node[shape=circle,draw] (1) at (0,1) {1};
  \node[shape=circle,draw] (2) at (1,1) {2};
  \node[shape=circle,draw] (3) at (2,1) {3};
  \node[shape=circle,draw] (4) at (2,0) {4};
  \draw (a) edge node {i} (b);
  \draw (a) edge node {ii} (c);
  \draw (b) edge node {iii} (c);
  \draw (1) edge node {i} (b);
  \draw (2) edge node {ii} (b);
  \draw (3) edge node {iii} (b);
  \draw (4) edge node {iii} (b);
\end{tikzpicture}
\end{array}
\]

we enumerate these pairs in \(Q_A \times 2^{Q_B}\),

\[(i, \{0\}), (ii, \{1,2\}), (ii, \{3\}), (iii, \{4\})\].

Note that we have to enumerate both \((ii, \{1,2\})\) and \((ii, \{3\})\). Since the final state \(iii\) of \(A\) is associated only with \(\{4\}\) which is a final state of \(B\), the containment holds.

Enumeration of subsets of states in \(B\) may, however, involve an explosion. As we will see later, this explosion is a serious problem when the algorithm is applied to tree automata. We use two techniques in order to suppress an explosion: (1) using ROBDD representation for counting subsets, and (2) recording not exact sets of subsets but their upward closures.

First, we reduce the representation of a set of subsets by using ROBDDs. The algorithm uses the following data structure.

\[D \in Q_A \mapsto \text{ROBDD}(Q_B)\]
Each entry $D(q)$ is a set including subsets of $Q_B$ associated with $q$. When we check the above containment problem, the contents of $D$ are as follows:

$$D(i) = \hat{0}$$
$$D(ii) = (\hat{1} \land \hat{2}) \lor \hat{3}$$
$$D(iii) = \hat{4}$$

where $\hat{x}$ denotes $(x, 0, 1)$.

Second, $D(q)$ does not record exact subsets that can be reached, but it also represents their arbitrary supersets. For example, $\llbracket D(i) \rrbracket = \llbracket \hat{0} \rrbracket$ represents all sets that subsume $\{0\}$, and $\llbracket D(ii) \rrbracket = \llbracket (\hat{1} \land \hat{2}) \lor \hat{3} \rrbracket$ represents sets that subsume either $\{1, 2\}$ or $\{3\}$. It is safe to do so because if there is $S$ in $\llbracket D(q) \rrbracket$ (e.g., $\{4\} \in \llbracket D(iii) \rrbracket$), the presence or absence of $S'$ such that $S \subseteq S'$ (e.g., $S' = \{0, 4\}$, $\{0, 1, 4\}$, etc.) does not affect the result of analysis. Thus, we can freely add $S'$ to $\llbracket D(q) \rrbracket$ if there is already $S$. More specifically, we can observe the following two properties.

- If $(r, T')$ is reached from $(q, S')$, there is $(r, T)$ reached from $(q, S)$ such that $T \subseteq T'$.
- For $q \in F_A$, if $S \cap F_B \neq \emptyset$, then so with $S'$, i.e., $S' \cap F_B \neq \emptyset$.

The first property states that when we compute $\llbracket D(r) \rrbracket$ from $\llbracket D(q) \rrbracket$, even if we add or ignore $S'$ in $\llbracket D(q) \rrbracket$ (e.g., $\{0, 1\}$ in $\llbracket D(i) \rrbracket$), we do neither gain nor lose information in $\llbracket D(r) \rrbracket$ up to its upward closures (e.g., using a label b, we can compute $\{3, 4\}$ for $\llbracket D(ii) \rrbracket$, but this is meaningless where there is $\{3\}$). The second property states that it is also safe to add or ignore $S'$ in the final step of the analysis (recall that in this step we check if all $S$ satisfy $S \cap F_B \neq \emptyset$). Therefore, the result of the containment check depends only on the subset $S$ but not on any upward subset $S'$.

Fortunately, an ROBDD nicely represents such an upward closure. A positive ROBDD corresponds to a boolean formula without negative occurrence of variables. Such an ROBDD is exactly what we want. The use of positive BDDs will further reduce the complexity of the analysis.

### 4.3 Algorithm

Given the data structure $D$, it remains to explain how to compute a transition for each entry in $D$ and propagate the result to the entry for the next state. The algorithm is not efficient if such a transition can only be done by explicitly counting elements in each $\llbracket D(q) \rrbracket$. In our algorithm, however, we rather compute transitions symbolically on each ROBDD representation.

We write $\delta(a, S, T)$ to denote a union of images $\bigcup \{\delta(a, q_1, q_2) \mid q_1 \in S, q_2 \in T\}$. Formally, we need a function that computes the set of unions of images $\{S \mid \delta(a, T, U) \subseteq S, T \in \llbracket \alpha \rrbracket, U \in \llbracket \beta \rrbracket\}$ of $\delta$ taking a triple of $a$ and two sets of sets, $\alpha$ and $\beta$, as arguments. This function is encoded by a simple function $tr$ in Figure 1 (which uses two ROBDD operations $\land$ and $\lor$ inside).

Using function $tr$, we compute $D$ as follows:
\[ tr \in \Sigma \times \text{ROBDD}(Q_B) \times \text{ROBDD}(Q_B) \rightarrow \text{ROBDD}(Q_B) \]
\[ tr' \in \Sigma \times Q_B \times \text{ROBDD}(Q_B) \rightarrow \text{ROBDD}(Q_B) \]
\[ tr'' \in \Sigma \times Q_B \times Q_B \rightarrow \text{ROBDD}(Q_B) \]

- Initialize \( D_0 \):
  \[ D_0(q) = \bigwedge_{q' \in I_B} (q', 0, 1) \quad (q \in I_A) \]
  \[ D_0(q) = 0 \quad \text{(otherwise)} \]  \hspace{1cm} (1)

- At \( i \)-th step, update \( D_i \) for each \( q_0, q_1, q_2 \in Q_A \) and \( a \in \Sigma \) such that \( q_0 \in \delta_A(a, q_1, q_2) \):
  \[ D_{i+1}(q_0) = D_i(q_0) \lor tr(a, D_i(q_1), D_i(q_2)) \]  \hspace{1cm} (2)

Repeat until \( D_i = D_{i+1} (= D) \) holds (fixpoint computation).

After all, we have

**Theorem 1.**

\[ [A] \subseteq [B] \Leftrightarrow \bigvee_{q \in F_A} D(q) \land \bigwedge_{q \in F_B} (q, 1, 0) = 0. \]

We check the right hand side of \( \Leftrightarrow \) using ROBDD operations. If it succeeds, the containment holds. The proof of the theorem can be found in Appendix B.

### 4.4 Experiments

We implemented the algorithm described so far. For comparison, we also implemented other algorithms in the literature.
**BDD** An ROBDD-based algorithm.

**DET** A textbook algorithm that involves determinization\(^4\).

**XDUCE** An XDuce’s algorithm by Hosoya, et al. [HVP00], which was originally implemented in Ocaml. We have re-implemented it in Java.

In implementation, we used our own ROBDD package in Java. We do not follow a particular variable ordering heuristics, but we rather used a random ordering. Investigation of variable ordering is left for future work.

Figure 2 shows the performance of each algorithm on the following examples:

**EX1** This example has an expression of the form \((a(\langle \rangle \mid b(\langle \rangle))^{15}a(\langle \rangle)(a(\langle \rangle \mid b(\langle \rangle))^{*}\) on the right hand side of containment. This regular expression is famous as its conversion to DFA from right to left leads to exponential blow up.

**EX2** The reverse of EX1. Determinizing \((a(\langle \rangle \mid b(\langle \rangle))^{*}a(\langle \rangle)(a(\langle \rangle \mid b(\langle \rangle))^{15}\) from left to right blows up states. The DET algorithm is not affected, which determinizes from right to left, while the XDUCE algorithm is affected.

**EX3** Another example that XDUCE cannot handle, including \(a(b_{1}(\langle \rangle))b_{1}(\langle \rangle)\ldots |a(b_{15}(\langle \rangle))b_{15}(\langle \rangle)\) on the right hand side.

**XHTML** A real-world example. We check that “xhtml1-transitional.dtd” contains “xhtml1-strict.dtd” [Wor00].

XDUCE is one of the best known algorithms but still causes blow-up as in EX2 and EX3. Our algorithm performs good in general.

---

\(^4\) It is known that if we restrict ourselves to DTDs, there is a more efficient algorithm for containment test. However this algorithm cannot be applied to other schema languages having the same expressive power as NTAs. Thus we here used a naive algorithm for containment of NTAs in general.
In some cases where the containment test fails, XDUCE can detect the problem very early. This is because XDUCE is a top-down algorithm, and problems are likely to be found near the roots of trees, i.e., more accurately, near the final states of tree automata. To simulate this early failure detection, we have to check if $D_i(q) = 1$ or not at each $i$-th step for each $q$ which is useful, i.e., there is a transition from $q$ that reaches a final state. Once there is such $q$, the test always fails.

5 Concluding Remark

This line of research aims at two major applications. One application is an XML schema version check tool based on the containment algorithm. As we noted earlier, the containment for some XML schemas such as RELAX NG [Ora01] is more difficult than what we have done in this paper. We are seeking the extension of our algorithm to do with these XML schemas. The other application is a type-checker of XML processing languages with types based on XML schemas. We are currently developing a typed XML processing language using the proposed NTA containment algorithm, which is released from IBM alphaWorks.

Acknowledgement

We would like to thank Bernd Finkbeiner for valuable discussions on related topics, and we also thank Makoto Murata for a lot of helpful suggestions.

References


In Figure 3, we summarize the standard operations on ROBDDs. Function $rd$ is called reducer functions which guarantee that resulting BDDs are reduced. Note that it is not efficient if these operations are implemented as is. In implementation, we use two kinds of hash tables (1) in order to use a unique pointer to refer to each syntactically equivalent BDD, and (2) in order to implement $\land$ and $\lor$ as memoise functions.

B  Proof

In this section, we describe the proof of the theorem in Section 4. The algorithm terminates as there are finitely many possibilities for $D_i$. Here we show its correctness.
\[ rd(x, \alpha, \beta) = \alpha \quad (\alpha = \beta) \]
\[ rd(x, \alpha, \beta) = (x, \alpha, \beta) \text{ (otherwise)} \]

\[ 0 \land \alpha = \alpha \land 0 = 0 \]
\[ 1 \land \alpha = \alpha \land 1 = \alpha \]
\[ \alpha \land \beta = \begin{cases} \text{rd}(\alpha.\text{var}, \alpha.l \land \beta, \alpha.h \land \beta) & (\alpha.\text{var} < \beta.\text{var}) \\ \text{rd}(\alpha.\text{var}, \alpha.l \land \beta.l, \alpha.h \land \beta.h) & (\alpha.\text{var} = \beta.\text{var}) \\ \text{rd}(\beta.\text{var}, \alpha \land \beta.l, \alpha \land \beta.h) & (\alpha.\text{var} > \beta.\text{var}) \end{cases} \]

\[ 0 \lor \alpha = \alpha \lor 0 = \alpha \]
\[ 1 \lor \alpha = \alpha \lor 1 = 1 \]
\[ \alpha \lor \beta = \begin{cases} \text{rd}(\alpha.\text{var}, \alpha.l \lor \beta, \alpha.h \lor \beta) & (\alpha.\text{var} < \beta.\text{var}) \\ \text{rd}(\alpha.\text{var}, \alpha.l \lor \beta.l, \alpha.h \lor \beta.h) & (\alpha.\text{var} = \beta.\text{var}) \\ \text{rd}(\beta.\text{var}, \alpha \lor \beta.l, \alpha \lor \beta.h) & (\alpha.\text{var} > \beta.\text{var}) \end{cases} \]

**Fig. 3.** BDD operations

In preparation, we define the notion that BDD \( \alpha \) is *positive* as follows.

\[ \forall S \in [\alpha]. \forall T. S \subseteq T \Rightarrow T \in [\alpha] \]

Assuming orderedness of \( \alpha \), we have \( \alpha \) positive, iff, either \( \alpha = 0 \), \( \alpha = 1 \), or \( \alpha = (x, \beta, \gamma) \) where \( \beta \) and \( \gamma \) are positive and satisfy \( [\beta] \subseteq [\gamma] \). Positiveness is closed under \( \land \) and \( \lor \), and thus all BDDs appearing in the computation of \( D \) are positive.

First, we need the following lemma. For any positive \( \alpha \) and \( \beta \),

\[ [[\text{tr}(a, \alpha, \beta)]] = \{ S \mid T \in [\alpha], U \in [\beta], \delta_B(a, T, U) \subseteq S \} \] (3)

where \( \delta_B(a, T, U) = \bigcup_{x \in T, y \in U} \delta_B(a, x, y) \). It is easy to see \( [[\text{tr}''(a, x, y)]] = \{ S \mid \delta_B(a, x, y) \subseteq S \} \), by using which we inductively show that \( [[\text{tr}'(a, x, \beta)]] = \{ S \mid U \in [\beta], \delta_B(a, x, U) \subseteq S \} \) for any positive \( \beta \):

**Case (\( \beta = 0 \)):** \( [[\text{tr}'(a, x, \beta)]] = \emptyset \).

**Case (\( \beta = 1 \)):** \( [[\text{tr}'(a, x, \beta)]] = 2^Q \). This is OK, since \( \emptyset \in [\beta] \).

**Case (otherwise):** By induction,

\[ [[\text{tr}'(a, x, \beta)]] = [[\text{tr}'(a, x, \beta.l) \lor (\text{tr}''(a, x, \beta.\text{var}) \land \text{tr}'(a, x, \beta.h))]] \]
\[ = \{ S \mid \exists U \in [[\beta.l]], \delta_B(a, x, U) \subseteq S \lor \exists T \in [[\beta.h]], \delta_B(a, x, \beta.\text{var}) \cup \delta_B(a, x, T) \subseteq S \} \]
\[ = \{ S \mid \exists U \in [[\beta]], \delta_B(a, x, U) \subseteq S \land \exists T \in [[\beta.h]], \delta_B(a, x, \beta.\text{var}) \cup \delta_B(a, x, T) \subseteq S \} \]
\[ = \{ S \mid \exists U \in [[\beta]], \delta_B(a, x, U) \subseteq S \} . \]
The last rewrite follows from positiveness of $\beta$. Proving (3) from the above result is a similar work.

Second, we prove that for any $q \in Q_A$,

$$[[D(q)]] = \{S \mid \exists v \in [q]_A. \{q' \mid v \in [q']_B\} \subseteq S\} \quad (4)$$

For (\textless{}), we show that $D(q)$ contains all possibilities. We prove that $\{q' \mid v \in [q']_B\} \subseteq [[D(q)]]$ for any $v \in [q]_A$ by induction on the definition of $[[q]_A]$.

**Case ($v = e$):** From (1), we have $I_B \subseteq [[D(q)]]$.

**Case ($v = a(v_1)v_2$):** There are $q_1$ and $q_2$ such that $q \in \delta(a, q_1, q_2)$ and $v_k \in [[q_k]]$ for $k = 1, 2$. The induction hypothesis guarantees $\{q' \mid v_k \in [q']_B\} \subseteq [[D(q_k)]]$ for any $k = 1, 2$. We use (3) to show that $\{q' \mid v_1 \in [q']_B, \{q' \mid v_2 \in [q']_B\} \subseteq [[D(q)]] \subseteq \{q' \mid a(v_1)v_2 \in [q']_B\}$.

For (\textless{}), we show that each $D_i(q)$ contains nothing too restrictive. We prove that for any $S \in [[D_i(q)]]$, there exists $v \in [q]_A$ such that $\{q' \mid v \in [q']_B\} \subseteq S$.

**Case ($i = 0$):** From (1).

**Case (otherwise):** For each $q_k$ and $S_k \subseteq [[D_{i-1}(q_k)]]$, let $v_k \in [[q_k]]$ be a witness of the induction hypothesis, i.e., $\{q' \mid v_k \in [q']_B\} \subseteq S_k$ for $k = 1, 2$. For any $S \in [[tr(a, D_{i-1}(q_1), D_{i-1}(q_2))]]$, (3) implies that there are $S_1$ and $S_2$ such that $\{q' \mid a(v_1)v_2 \in [q']_B\} \subseteq \delta_B(a, S_1, S_2) \subseteq S$. Therefore any $S$ added to $D_i(q)$ by step (2) (where $q \in \delta_\alpha(a, q_1, q_2)$) has a new witness $a(v_1)v_2 \in [q']_A$.

Finally we prove Theorem 1 as follows.

$$\forall q \in F_A. D(q) \land \land_{q \in F_B} (q, 1, 0) = 0$$

$$\Leftrightarrow \forall q \in F_A. \{S \mid \exists q \in F_A. S \in D(q)\} \cap \{S \mid F_B \cap S = \emptyset\} = \emptyset$$

$$\Leftrightarrow \forall q \in F_A. \forall S \subseteq D(q). F_B \cap S = \emptyset$$

We use (4) and obtain

$$\Leftrightarrow \forall q \in F_A. \forall v \in [q]_A. \forall S. \{q' \mid v \in [q']_B\} \subseteq S$$

$$\Rightarrow F_B \cap S = \emptyset$$

$$\Leftrightarrow \forall q \in F_A. \forall v \in [q]_A. F_B \cap \{q' \mid v \in [q']_B\} = \emptyset$$

$$\Leftrightarrow [A] \subseteq [B].$$

\[\square\]
Weak Minimization of DFA – An Algorithm and Applications

Bala Ravikumar
Department of Computer Science, Sonoma State University
Rohnert Park, CA 94128
ravi@cs.sonoma.edu

Abstract. DFA minimization is a central problem in algorithm design and is based on the notion of DFA equivalence: Two DFA’s are equivalent if and only if they accept the same set of strings. In this paper, we propose a new notion of DFA equivalence (that we call weak-equivalence): We say that two DFA’s are weakly equivalent if they both accept the same number of strings of length \( k \) for every \( k \). The motivation for this problem is as follows. A large number of counting problems can be solved by encoding the combinatorial objects we want to count as strings over a finite alphabet. If the collection of encoded strings is accepted by a DFA, then standard algorithms from computational linear algebra can be used to solve the counting problem efficiently. When applying this approach to large-scale applications, the bottleneck is the space complexity since the computation involves a matrix of order \( k \times k \) if \( k \) is the size of the underlying DFA \( M \). This leads to the natural question: Is there a smaller DFA that is weakly equivalent to \( M \)? We present an algorithm of time complexity \( O(k^2) \) to find a compact DFA equivalent to a given DFA. We illustrate, in the case of tiling problem, that our algorithm reduces a (strongly minimal) DFA by a factor close to 2.

1 Introduction

In this paper, we explore an application of finite automata to counting problems. To count the number of combinatorial structures of a certain size \( n \), we map each structure to a string \( s \) from a carefully chosen alphabet \( S \) so that there is a 1-1 mapping between each object we are counting and a subset of strings over \( S \). When establishing this 1-1 mapping, we can ignore the size parameter \( n \), taking care only that it is length-preserving. If the set of such encoded strings is regular, then we have arrived at an efficient algorithm to solve our counting problem. This is a consequence of the fact that there is an efficient algorithm for counting the number of strings of length \( n \) accepted by a DFA.

We begin with an illustration of this approach to solve the following counting problem. Consider the following problem taken from Liu’s book [Liu]: ”Find the number of \( n \)-bit binary sequences that have the pattern 010 occurring for the first time at the \( n \)-th digit.” For this problem, the encoding is trivial. We simply use the string itself as the encoding. The set of strings that have a unique occurrence
of 010 as suffix is regular and can be accepted by the DFA shown in the following figure.

![DFA Diagram](image)

**Fig. 1.** DFA for $L = \{x \mid x$ ends with a unique occurrence of 010 as a suffix}$

Let $M$ be a DFA with $k$ states. The transition matrix $A$ of the DFA is a $k \times k$ matrix where $a_{ij}$ is the number of transitions from state $i$ to state $j$. It is well-known and is easy to show by induction on the length that the number of strings of length $n$ that start in state $i$ and end in state $j$ is given by $[A^n]_{ij}$. Thus, the number of strings of length $n$ accepted by DFA $M$ is given by $xA^n y'$ where $x$ is the start vector (of order $1 \times k$) such that $x_j = 1$ if and only if $j$ is the start state. $y'$ is the transpose of $y$. $y$ is the accepting vector (of order $1 \times k$) where $y_j = 1$ if and only if $j$ is an accepting state. For the above problem, it is easy to verify that $A =$

$$
\begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

$x = (1 \ 0 \ 0 \ 0)$ and $y =$

$$
\begin{bmatrix}
0 \\
0 \\
0 \\
1
\end{bmatrix}
$$

Note that we have removed the dead state since it does not contribute to the number of strings accepted by the DFA.

The above technique to solve counting problems is sometimes called the transfer matrix method [Wil]. A DFA based approach can be very effective in creating the transfer matrix. For example, the closure properties of regular languages and other nice structural features can be used to design the DFA systematically. What motivated this work is the fact that the theory of finite automata can help
in reducing the size of the transfer matrix as well. Optimizations such as removing unreachable states, identifying symmetries, minimization of the DFA (in the classical sense) are some of the ways to reduce the size of the transfer matrix. However, there is room for further improvement. Since we are only interested in using the DFA for counting the number of strings of given length, we need not limit as candidates (in the minimization) only the DFA’s that accept the same language. We can change the language so long as we don’t change the number of strings of every possible length.

This leads to the notion of weak equivalence of DFA’s: We say that two DFA’s are weakly equivalent if they both accept the same number of strings of length \( k \) for every \( k \). The main goal of this paper is to describe an algorithm that finds a DFA \( M' \) with fewer states that is weakly equivalent to a given DFA \( M \). In this preliminary version, we will focus on the algorithm for weak minimization but describe the applications only briefly, namely counting domino tilings and self-avoiding walks.

The rest of the paper is organized as follows. In Section 2, we describe an implementation that takes as input an integer \( k \) and automatically creates a DFA that accepts encodings of valid tilings of the \( k \times n \) checkerboard (for all \( n \)). We also describe an implementation for automatically synthesizing a DFA for self-avoiding walks in the lattice grids of a specified width. In Section 3, we consider the time complexity of solving the counting problem after the transfer matrix has been created. In Section 4, we present the main result of the paper - an algorithm for weak minimization and prove its correctness. We also estimate its time complexity, along with some implementation details. In Section 5, we state some open problems and directions for future work.

2 DFA Design for Counting Problems

In this section, we are concerned with the design of DFA for two counting problems - tiling and self-avoiding walks. Both problems are parameterized by \( k \), the number of columns. These and other counting problems require that software tools be developed to create DFA automatically from given specifications. We are currently in the process of creating a software engineering framework for such a general purpose tool. However, in this section, what we describe is a more focussed and narrowly defined effort to create programs tailor-made for the two specific applications. In Section 2.1, we will describe an algorithm to design DFA that accepts encoded tilings, and in Section 2.2, we describe a similar algorithm for self-avoiding walks on a rectangular grid. In both cases, the width is a parameter provided as input to the program.

2.1 DFA Design for Tiling Problem

We are interested in counting the number of ways to tile a \( k \) by \( n \) checker-board with some removed squares. This problem is identical to counting the number of perfect-matchings of a subgraph \( G \) of the grid graph of order \( k \) by \( n \). The
connection between the two problems is as follows: We can create a graph from
the checker-board by making each unremoved square a vertex, and connecting
two vertices by an edge if the two squares share a common edge. It is clear that
a 1 × 2 or a 2 × 1 tile corresponds to an edge, and thus a set of tiles corresponds
to a matching. A valid tiling thus corresponds to a perfect matching. Counting
perfect-matchings in a bipartite graph is known to be #P-complete [Val]. Even
when the instance is restricted to a subgraph of a grid graph, the problem seems
to be hard, as recently shown by Ogihara and Toda [Ogi]. The tiling problem
with the number of columns restricted to a constant has also been extensively
studied, see, for example [Kla, Sta, Pro, Pac], among others.

The encoding of tiling can be illustrated with k = 3. We represent the 1 × 2
horizontal tile by the string [2 2], and the 2 × 1 vertical tile by [1 1]T.

Thus our encoding will use the alphabet Σ = {a, b, c} where
a = [1 1 2]T, b = [2 1 1]T and c = [2 2 2]T. The DFA M that accepts valid tiling encoded
using the above alphabet is shown in the figure below.

A DFA design for general k is based on enumerating the reachable states
from the start state as follows: a state is represented by a string of length k over
(0, 1) in which a 1 in position j indicates that the first half of a horizontal tile
(represented by symbol 2) was seen in position j in the last input symbol. This
enforces the requirement that the j-th component in the next input must be a 2
for the tiling to be valid.

From this argument, it should be clear that the set of reachable states is
as follows. (0, 0, ..., 0) is reachable and is the start state as well as the unique
accepting state. If (x1, x2, ..., xk) is reachable from (0, 0, 0, ..., 0), then so is (y1, y2,
... ,yk) where yj = 0 if xj = 1, and for any j such that xj = 0 and xj+1 = 0,
either (yj = 0 and yj+1 = 0) or (yj = 1 and yj+1 = 1). Transitions are as follows:
δ ((x1, x2, ..., xk), (t1, t2, ..., tk)) = (f(x1, t1), ..., f(xk, tk)) where f(0, 1) = 0,
f(1, 2) = 0 and f(0, 2) = 1. It is straight-forward to implement a program that
given an integer k, generates the DFA M(k) described above.

![Fig. 2. A tiling DFA](image-url)
This DFA can be used to find the number of perfect-matchings in the \( m \times n \) grid graph. But we are interested in a more general problem, namely to count the number of perfect-matchings in a subgraph of the grid graph. We can handle this more general problem as follows: The input to this problem is integer \( k \), followed by a sequence \( I \) of squares (denoted by the (row number, column number) pair) that have been removed from the board. First, we design a DFA over a \( k \)-column alphabet in which each entry is \( \{0, 1, 2\} \) as described above. Note that \( I \) also has information about the number of rows \( r \) in the board. 1 and 2 have the same meaning as above, but 0 now represents a removed square. Our program creates another DFA \( M(I, r) \) that accepts all the strings of length \( r \) that have 0 in row \( j \) and column \( k \) if and only if the square \((j,k)\) is in \( I \). (There is no other restriction placed on the accepted strings.) Clearly, \( M(I, r) \) has size at most \( O(r \times 2^k) \). The number of perfect matchings in the grid graph associated with \( I \) (as the removed squares) is the number of strings of length \( r \) in the language \( L(M(I, r) \cap L(M(k))) \). We use the standard pair construction to obtain a DFA that accepts this language and use the transfer matrix to count the number of perfect-matchings. It is clear from the above discussion that significant optimizations should be done to speed up the computation since the DFA’s generated automatically through a series of mechanical steps tend to be rather large.

2.2 DFA Design for Self-Avoiding Walks in a \( k \times n \) Grid

Design of DFA to accept the encodings of self-avoiding walks in a \( k \times n \) rectangular grid is more complex than the tiling problem described above. We describe our encoding scheme using \( k = 2 \) (namely paths in \( 2 \times n \) grid) as strings over a finite alphabet. We split the path along the horizontal grid lines, and each segment of the walk can be encoded as a single symbol. This leads to the following 12 symbols shown in the figure below. It is not hard to design a DFA that reads a series of input symbols over the above listed alphabet and checks that the input is valid, namely the input string represents a simple path. This requires maintaining the position of one or both ends of the path as we scan it from left to right. For \( k = 2 \), the DFA is shown in table 1.

It is not easy to compute the number of simple paths of length \( n \) directly from the above DFA. The reason is that the length of the encoded string does not
not represent the length of the path since the symbols encode variable number of edges. (For example, the left-most symbol in Figure 3 encodes 3 edges, while the right-most one encodes 1 edge.) So we map the strings from the above alphabet to another alphabet so that a simple path of length $n$ will now encoded by a string of length $n$ using this new alphabet. For $k = 2$, this new alphabet has 7 symbols. In general, this new alphabet will have $3k + 1$ symbols in it.

The idea is as follows: we encode each edge in the walk by one of the four symbols $1 \rightarrow \text{West}$, $2 \rightarrow \text{North}$, $3 \rightarrow \text{South}$ and $4 \rightarrow \text{East}$. We then map each symbol in the alphabet of Figure 4 into a sequence of symbols that denote the actual edges used. For example, the left-most symbol in Figure 3 will be denoted by $123$. Our final design of DFA uses a two-step approach: First we design a DFA as shown in Table 1. Then we transform this DFA into one that encodes the paths using symbols $1, 2, 3, 4$ etc. Since the latter encoding is length-preserving, we can apply the matrix power formula on the transfer matrix derived from the latter DFA.

$$M = < Q, S, \delta, 1, \{1, 2, 3, \ldots, 13\} >.$$ $S$ consists of the 12 symbols $\{1, 2, 3, \ldots, 12\}$ in the order in which they appear in Figure 3. The transition function $\delta$ is as follows.

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>12</td>
<td>4</td>
<td>11</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

The missing transitions are to a dead state that is not shown. We have used some programming tools to generate the DFA that accepts simple paths in a grid with $k = 3$ columns. One of our eventual goals is to count the number of self-avoiding walks in $k \times k$ board for as large a value as possible. The current record is $k = 51$ [Mad]. The program we have developed to generate self-avoiding walk for $k = 3$ is not general enough to deal with arbitrary $k$. Many programming challenges must be overcome in order to design a DFA for arbitrary $k$. When
this is done, one can consider the question of whether \( k = 51 \) can be surpassed using this approach.

3 Some Computational Issues

Counting the number of strings of length \( n \) accepted by a DFA raises many interesting computational problems. A direct approach would focus on the most efficient way to compute the matrix expression \( xA^n y' \) where \( A \) is a \( k \times k \) matrix of integers, \( x \) and \( y \) are \( k \)-vectors. Let \( T(k) \) be the number of arithmetic operations (additions and multiplications) sufficient to compute the product of two \( k \times k \) matrices. Coppersmith and Winograd’s algorithm [Cop] establishes that \( T(k) = O(k^a) \) where \( a \) is approximately 2.4. Thus the arithmetic complexity of computing \( xA^n y' \) is at most \( \min \{ O(k^a \log n), O(k^2n) \} \) based on two ways to compute the product. To get the first bound, we could compute \( A^n \) using repeated squaring with the fast matrix multiplication algorithm as the base case, then pre- and post-multiply by vectors \( x \) and \( y' \). To get the latter bound, we simply multiply the matrices from left to right. There are \( n + 1 \) multiplications involved. Each of them (except the last one) requires \( O(k^2) \) operations since it involves multiplying a \( k \)-vector by a \( k \times k \) matrix. The final multiplication is a dot product that takes \( O(k) \) steps. The total cost is clearly seen as \( O(k^2n) \).

By combining the two algorithms, we can get the following upper-bound result.

**Theorem 1.** The number of integer multiplications required to compute \( xA^n y' \) (where \( x \) is a \( 1 \) by \( k \), \( A \) is \( k \times k \), and \( c \) is a \( k \times 1 \) vector) is at most \( \min_{0 \leq r \leq \lfloor \log n \rfloor} \{ 2rk^3 + k^2(n - 2^r) \} \)

This bound is obtained as follows: First compute \( A^{2^r} \) by repeated squaring \( r \) times. This involves \( 2rk^3 \) multiplications. The product \( xA^{2^r} A^{n-2^r} y' \) can be computed by left to right matrix multiplication with at most \( k^2(n - 2^r) \) scalar operations. Since we can choose \( r \), the bound follows.

We have implemented the above algorithm where the matrix entries can be arbitrarily long integers. We have been able to solve instances for which DFA’s have several 1000 states and \( n \) (the length of the string) can also be in thousands.

An entirely different approach to computing \( xA^n y' \) is based on inverting a Toeplitz matrix. The basic idea behind this method is as follows: if \( f(n) = xA^n y' \) where \( A \) is a \( k \times k \) matrix, then \( f(n) \) satisfies a linear recurrence equation with constant coefficients [Ste]. We can compute \( f(n) \) for any \( n \) by determining the coefficients in the linear recurrence formula. This can be accomplished efficiently by inverting a Toeplitz matrix. We have also implemented this algorithm. In the final version, we will provide a comparison between different ways to count the number of strings of a given length accepted by a DFA in different applications.

4 Weak Minimization Algorithm

From the discussion above, it is clear that reducing the number of states in the DFA is crucial for the success of DFA based approach for counting problems. The
standard optimizations we should first attempt are: (1) remove useless states and (2) perform DFA minimization (in the classical sense). The following example illustrates how we can optimize beyond standard DFA minimization.

**Example:** Consider the DFA $M$ shown in the following figure. It is a minimal DFA, but we will show below that $M'$ is a smaller DFA that is weakly equivalent to $M$.

![DFA Diagram](image)

**Fig. 4.** $M$ (on the left) is strongly minimal, but not weakly minimal

If we switch the transitions on input symbols 0 and 1 from state $B$ (an operation that preserves weak equivalence) in $M$, the resulting DFA is not minimal and by (classical) minimization of this DFA, we get $M'$.

The above idea leads to the following concepts and problems. Define two DFA’s to be weakly equivalent if the number of strings of length $n$ accepted by the two DFA’s are the same for all values of $n$. This leads to the following computational problems.

**Weak equivalence problem**

*Input:* Two DFAs $M_1$ and $M_2$.

*Output:* yes if $M_1$ and $M_2$ are weakly equivalent, no else.

**Weak Minimization Problem**

*Input:* A DFA $M$.

*Output:* A DFA $M'$ with as few states as possible such that $M$ and $M'$ are weakly equivalent.

Due to space limitation, we will omit discussion of weak equivalence problem.

### 4.1 Algorithm for Weak Minimization

The basic idea behind this algorithm is as follows. Recall the Myhill-Nerode theorem [Hop] that states that a DFA $M$ is minimal if and only if (a) $M$ has no
unreachable states (from starting state) and (b) for any two states \( p \) and \( q \), there is some string \( x \) such that exactly one of \( \delta(p, x) \), \( \delta(q, x) \) is in \( F \). A simple (strong) minimization algorithm is: "Create an initial partition of vertices into \( S_1 = F \) and \( S_2 = Q \setminus F \). If there is a pair of states \( p, q \in S_j \), and an input symbol \( a \in \Sigma \) such that \( \delta(p, x) \) and \( \delta(q, x) \) are in different \( S_j \)'s, split \( S_j \) such that all states that go to the same \( S_k \) on input \( a \) are in the same partition of \( S_k \). This process of splitting continues until no split is possible. At this stage, each group contains states that are equivalent to each other and hence they can be merged to form the minimal DFA."

We use a similar idea. We associate with each equivalence class \( S_j \), a vector \( v \) of length \( k \) for some \( k \). For a vector \( v \), we denote the \( j \)-th component as \( v[j] \). This vector represents the number of strings of length \( i = 0, 1, 2, ..., k \) that are accepted by the DFA, starting from any state in that class. At any stage, states in the equivalence class \( S_j \) are such that for all \( k = 0, 1, ..., |v| \) (where \( v \) is the vector associated with \( S_j \)), the number of accepting computations starting from each state in \( S_j \) is the same number, namely \( v[k] \). Define two states \( p, q \in Q \) as weakly equivalent if for all \( k \), the number of accepting strings of length \( k \) starting from \( p \) is the same as the number of accepting strings of length \( k \) starting from \( q \).

A key fact we will use is the following:

**Theorem 2.** Let \( M = < Q, S, \delta, s, F > \) be a DFA with \( m \) states, and let \( p, q \) be two states in \( Q \). If the number of strings of every length \( k = 0, 1, 2, ..., m-1 \), starting from \( p \) and \( q \) is the same, then \( p \) and \( q \) are weakly equivalent.

**Proof:** Let \( M \) be a DFA with \( m \) states. For a state \( p \in Q \), define \( f_M(n, p) \) as the number of strings of length \( n \) that leads to acceptance starting from \( p \). Clearly, \( f_M(n, p) = r_p A^n y \) where \( A \) is the transfer matrix associated with \( M \), \( y \) is the accepting vector and \( r_p \) is the start vector with a 1 in position \( p \), and 0 in all other positions. It can be shown that \( f_M(n, p) \) satisfies a linear recurrence equation of order \( m \). Thus, \( f_M(n, p) \) and \( f_M(n, q) \) both satisfy linear recurrence equations of order \( m \). Let \( f_M(t, p) = \sum_{i=1}^{m} a_i f_M(t-i, p) \).

Similarly let \( f_M(t, q) \) be given by: \( f_M(t, q) = \sum_{i=1}^{m} b_i f_M(t-i, q) \).

We are given that \( f_M(n, p) = f_M(n, q) \) for all \( n = 0, 1, ..., m-1 \). From equation 1 for \( f_M(t, p) \), we can write down a system of linear equations by taking \( t = m, m+1, ..., 2m-1 \). Similarly, we can write down a system of \( m \) equations in \( m \) unknowns for \( f_M(t, q) \). We have two systems of linear equations with \( a_j \)'s and \( b_j \)'s as the unknown. The equations have identical LHS as well as RHS coefficients, so they must have the same solutions. Thus, \( a_j = b_j \) for all \( j \). This concludes the proof.

The idea behind the algorithm is to compute in an incremental manner the number of accepting strings of length \( k \) starting from each state and maintain an equivalence class that includes all the states that have the same count for all \( j = 0, 1, ..., k \). In other words, two states are \( k \)-weakly equivalent if \( f_M(j, p) = f_M(j, q) \) for all \( j \leq k \). We refine the partition during the next iteration based on \( (k+1) \)-weak equivalence, just as in the strong-minimization algorithm. The algorithm terminates after \( m \) iterations. Now we can merge the equivalence classes into a single state. A more formal description of the algorithm follows.
Algorithm WeakMinimizeDFA
Input: DFA $M = \langle Q, S, \delta, s, F \rangle$
// Assume that M is strongly minimal DFA with useless states removed.
1. $S = \{S_1, S_2\}$ where $S_1 = F$ and $S_2 = Q \setminus F$.
Assocate vector $v_1 = <1>$ and $v_2 = <0>$ with the two collections.
// $v_1$ represents the number of strings of length 0 accepted starting
// from a state in $F$ and $v_2$ is similar.
2. $t = 0; n = 2; // n is the size of $S$.
3. repeat $|Q|$ times
   { for every $k$ do
      for every $i \in S_k$ do
      { Initialize a vector $c$ of length $|S_k|$ to 0;
        for each input $a$ in $S$ do
        3.1. next = $\delta(i, a)$;
        3.2. $s[i] = s[i] + v_p(t)$ where $S_p$ is the class to which next belongs;
        endfor;
        if (not all the entries in $s$ are the same) then
        { partition $S_k$ into equivalence classes according to the equivalence relation
          $p \equiv q$ if and only if
          $s[p] = s[q]$ and name them $S_k, S_{n+1}, ..., S_{n+r}$ and set $n = n + r$;
        }
      }
    }
4. for each $j$, merge the states in $S_i$ into a single state $j$ of a new DFA $M'$.
Define the transitions of $M'$ as follows: pick any state in $S_i$. Let $m$ be the
number of transitions from $S_i$ to $S_k$. Add $k$ directed edges from $i$ to $k$.
Do this for every $S_i$. Now there must be exactly $|S|$ out edges from each $i$.
Arbitrarily label these edges with symbols from $S$. For each state $i$, if every state
in $S_i$ is an accepting state, make $i$ an accepting state in $M'$; else it will not be an accepting state.
5. output $(M')$ and halt.

4.2 Correctness of the Algorithm and Its Time Complexity
We claim that $M'$ is weakly equivalent to $M$. This can be shown as a direct
consequence of the above theorem since we are computing for all states $p$, the set
of strings of length $k$ accepted starting from $p$. If the two vectors are identical
after $k$ iterations, it follows from the above theorem that the two states are
weakly equivalent. Thus, we can combine them into a single state as done in
Step 4. It is clear that $M'$ has no more states than $M$ and in practice, the
algorithm reduces the number of states significantly. In fact, we do not know
any instance in which the algorithm produces a sub-optimal DFA. But one of
the reasons for our inability to produce such counter-examples is that we don’t
know how to construct weakly optimal DFA. However, we approached this issue
indirectly as follows: Suppose we start with a DFA $M$ and strong-minimize it to
get $M'$. We send both $M$ and $M'$ as inputs to our weak minimization algorithm. Suppose our algorithm produces two DFA’s with different number of states, we would produced a counter-example. So far, we have tried this approach on some examples, but in each case, the output produced had the same number of states.

On the other hand, it may be the case that the algorithm always produces a weakly optimal DFA. But the proof of optimality of strong minimization can’t be directly extended to our algorithm. One of the reasons is that there are some basic differences between strong minimization and weak minimization. For example, unlike the fact that (strongly) minimal DFA is unique up to isomorphism, weakly minimal DFA are not unique.

**Time complexity:** Let $n$ be the number of states in the DFA $M$ and $k$ be the number of input symbols. Steps 1 and 2 require $O(n)$ steps. The algorithm performs $n$ iterations of Step 3. In each step, we extend the length each of the $n$ vectors by performing an integer addition. Then, we examine the $j$-th component of the vector associated with each state in an equivalence class and split. It is clear that the cost of all these operations is at most $O(n)$. Thus the total cost of Step 3 is $O(n)$. The cost of Step 4 is easily seen to be $O(kn)$. Thus the total time complexity is $O((k + n)n)$. In fact, since we are interested in counting, we can skip Step 4 and proceed directly to the create the transfer matrix in which case, the total complexity is $O(n^2)$. Thus we have:

**Theorem 3.** The time complexity of creating an optimized transfer matrix from a given DFA using the algorithm presented above is $O(n^2)$.

**Experimental Results:** We have implemented the above algorithm and have tested it on the tiling DFA’s described in Section 2.1. The following table shows the size of the strongly minimized DFA from the DFA generated the by program of Section 2.1 and the size of the weak-reduced DFA (based on the algorithm presented above) for various values of $k$.

<table>
<thead>
<tr>
<th>$k$ (column width)</th>
<th>strong-minimal DFA</th>
<th>weak-reduced DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>70</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>252</td>
<td>114</td>
</tr>
<tr>
<td>10</td>
<td>252</td>
<td>142</td>
</tr>
</tbody>
</table>
5 Summary of Contributions and Directions for Further Work

We have accomplished the following goals in this paper: (1) We argued that a number of counting problems can be solved in a unified manner using a DFA based approach. As a case study, we showed that DFA based approach can be used to count the number of simple paths in a grid graph and the number of ways to tile a lattice grid. (2) Optimization issues arising from our approach led us to propose a new notion called DFA weak minimization. (3) We designed and implemented an efficient algorithm to weak-minimize a DFA.

This study has raised several interesting practical and theoretical problems. Here is a short list of them: (1) Prove or disprove: The algorithm presented in Section 4 always finds a weakly minimal DFA. (2) If the answer to question (1) is negative, what is the complexity of weak minimization? Is it solvable in polynomial time? (3) Develop a software design framework that converts a DFA specification into a DFA realization. Implicit representations and other compact representations can be pursued in counting applications to surmount the storage requirements of transfer matrix. (4) It is obvious that there are non-regular languages that have the same counting function as regular languages. A systematic identification of such languages will extend the scope of counting problems that can be solved using transfer matrix approach.

Acknowledgment

Many of the programs described above were implemented by my graduate student Xiaoming Lu.

References


Bag Automata and Stochastic Retrieval of Biomolecules in Solution

Mark Daley¹, Mark Eramian¹, and Ian McQuillan²

¹ Department of Computer Science, University of Saskatchewan
Saskatoon, Saskatchewan, S7N 5A9, Canada
{daley, eramian}@cs.usask.ca

² Department of Computer Science, University of Western Ontario
London, Ontario, N6A 5B7, Canada
imquill@csd.uwo.ca

Abstract. In this paper, we define the notion of a well-formed bag automaton, which is used to model the inherent problems associated with the retrieval of biomolecules from solution. We will consider the computational power and other properties of non-deterministic and deterministic, well-formed bag automata.

1 Introduction

One of the most exciting areas of current investigation in computer science is the field of biocomputing. The ability to harness biological molecules and biochemical mechanisms to perform arbitrary computations and store data offers us opportunities for parallel computation on a hitherto impossible scale.

Many models of biocomputing rely upon the used biochemicals being in solution. This introduces an interesting problem, as interactions between chemicals in solution is a purely stochastic process. We are concerned here specifically with the case of data storage using biomolecules in solution. Clearly, if data objects are allowed to freely move throughout a solution, we cannot have an “addressable” memory, at least in the sense we are accustomed to with electronic computers. The best that can be done is to include in each data item a tag indicating what its address is.

The ensuing difficulty is that finding a particular address becomes as difficult as finding the data item itself. Since the data items in this case are biomolecules in solution, they are simple unordered collections. There exist techniques (e.g. magnetic-bead separation) for extracting biomolecules with specific “tags”, however these techniques are extremely time-consuming and difficult to automate thus leaving them impractical for use in a large system. We believe it makes sense, then, to consider the capabilities of a biocomputing system with a purely non-addressable storage.
In order to gain some insight into the formal properties of computers based on non-addressable data stores, we need to investigate computational models with non-deterministic storage. For example, a pushdown is a deterministic storage type, as given some configuration, when we read some symbol from the pushdown store, it is always unique. The classical computer science literature is filled with many models of computation, but in almost all cases, the storage method used in these models is deterministic. An interesting exception is the use of associative memory [2], which allows for retrieval of data by content rather than by address. One can see the relevance of studying various computing models with non-deterministic storage to both biocomputing and bioinformatics.

In this paper we will introduce a new formal model of data storage that involves a non-deterministic storage type called a bag which can hold symbols from a finite alphabet. Further, we define a well-formed bag automaton whereby at each time step, one can examine some symbol in the bag and either remove it, leave it stationary or toss an additional symbol into the bag (in addition to leaving the symbol examined stationary). Notice that any symbol that is present in the bag can be examined at each time step. However, with this model, every input word leading to acceptance must always get accepted regardless of the symbol we examine at each step. Indeed, this is a relevant notion as it is most difficult to find a particular “symbol” in solution, as mentioned above. In addition, we consider deterministic and non-deterministic transition functions for well-formed bag automata.

We discover that the computational power of these machines is quite limited. We show that the deterministic variant accepts a strict, infinite hierarchy (with respect to the size of the bag alphabet) between the deterministic counter and deterministic pushdown languages. Similarly, the nondeterministic bag languages are between the nondeterministic counter and context-free languages. This result lends evidence to support the necessity of using magnetic-bead separation or other such techniques in single pot computations. Indeed, there must be some method of retrieving the desired biomolecules.

We begin with a brief review of relevant notions.

2 Preliminaries

Let $\mathbb{N}$ be the set of positive integers and let $\mathbb{N}_0$ be the set of non-negative integers.

We refer the reader to [6] for language and automata theory preliminaries. An alphabet $\Sigma$ is a finite set of symbols. We denote the empty word by $\lambda$. We let $\Sigma^*$ and $\Sigma^+$ be the sets of all words and of all non-empty words, respectively, over $\Sigma$. A language, or $\lambda$-free language, over $\Sigma$ is some subset of $\Sigma^*$, or $\Sigma^+$, respectively. Let $w \in \Sigma^*$. We let $\text{alph}(w)$ be the set of all letters of $w$. We denote by $w^R$ the word obtained from $w$ by reversing the letters. We denote by $|w|$ the length of $w$ and by $|w|_a$, $a \in \Sigma$, the number of occurrences of $a$ in $w$. We denote by $\text{perm}(w)$, the set of words obtained by permuting the positions of the letters of $w$. 

We denote the families of pushdown automata (respectively context-free languages) by \(\text{pda (CF)}\), one-counter automata (respectively one-counter languages) by \(\text{nca (NCL)}\), deterministic pushdown automata (respectively deterministic context-free languages) by \(\text{dpda (DCF)}\), deterministic one-counter automata (respectively deterministic one-counter languages) by \(\text{dca (DCL)}\) and recursively enumerable languages by \(\text{RE}\).

Each of the one counter automata above is considered to be a pushdown automaton with only one pushdown symbol in addition to an end-marker \(Z_0\), accepting by final state.

3 Bag Automata

We define a bag storage type as a tuple
\[
\Omega = (\Gamma, C, I),
\]
where \(\Gamma\) is a finite alphabet called the bag alphabet and \(C\) is the set of mappings \(r\) from \(\Gamma\) into \(\mathbb{N}_0\) where values of \(r\) are denoted by \(r(x), x \in \Gamma\), and \(r\) is written as \(r = \sum_{x \in \Gamma} r(x)x\). We call each element of \(C\) a bag and \(C\) the set of bags. Alternatively, a multiset could be used to represent each bag. Multiset automata, \(\text{[3]}\), contrast bag automata in that they accept input multisets.

For \(r \in C\) we denote by \(\text{supp}(r) = \{x \mid r(x) \neq 0\}\) the support of \(r\). The series where \(r(x) = 0\) for all \(x \in \Gamma\) is written as 0. Traditionally, series with finite support are called polynomials and so all bags considered in this paper are polynomials. For an arbitrary \(x \in \Gamma\), we denote by \(nx\) (respectively \(x\)), the bag \(r\) such that \(r(x) = n\) (respectively \(r(x) = 1\)) and \(r(y) = 0\) for all \(y \neq x\). These are traditionally called monomials.

Let \(r = \sum_{x \in \Gamma} r(x)x\) and \(r' = \sum_{x \in \Gamma} r'(x)x\) be in \(C\). We say \(r \leq r'\) if and only if \(r(x) \leq r'(x)\) for all \(x \in \Gamma\). The size of \(r\) is \(\sum_{x \in \Gamma} r(x)\).

We denote the sum of two bags \(r\) and \(r'\) as \(r + r' = \sum_{x \in \Gamma} (r(x) + r'(x))x\). Also, we denote the difference of bags \(r\) and \(r'\) as \(r - r' = \sum_{x \in \Gamma} (r(x) - r'(x))x\) which is defined if and only if \(r' \leq r\).

The set \(I = \{\text{toss}(x), \text{grab}(x), \text{stay} \mid x \in \Gamma\}\) are partial functions from \(C\) to \(C\) defined such that \(\text{toss}(x)(r) = r + x, \text{stay}(r) = r\) and \(\text{grab}(x)(r) = r - x\) defined if and only if \(r - x\) is defined. We call \(I\) the set of bag instructions. Notice that \(r + x\) is the sum of a polynomial and a monomial.

We now define bag automata, after which we will consider a well-formed variant by placing a restriction on our original definition.

We will denote \(I_0 = \Gamma \cup \{0\}\) in the sequel.

A one-way, non-deterministic bag automaton is a 6-tuple,
\[
M = (\Omega, Q, \Sigma, \delta, q_0, F),
\]
where \(\Omega = (\Gamma, C, I)\) is a bag storage type, \(Q\) is the set of states, \(\Sigma\) is the input alphabet, \(q_0 \in Q\) is the initial state and \(F \subseteq Q\) is the set of final states. The transition mapping \(\delta\) is from \(Q \times (\Sigma \cup \{\lambda\}) \times I_0\) into finite subsets of \(Q \times I\).
such that both $x = y$ and $x \neq 0$ must hold whenever $(q, \text{grab}(y)) \in \delta(p, a, x)$, for $p, q \in Q, a \in \Sigma \cup \{\lambda\}, x, y \in \Gamma_0$.

We say a bag automaton $M = (\Omega, Q, \Sigma, \delta, q_0, F)$ is deterministic if for all $q \in Q$ and $y \in \Gamma_0$, $\delta(q, a, y) \cup \delta(q, \lambda, y)$ is the empty set or a singleton for each $a \in \Sigma$.

A configuration of a bag automaton $M = (\Omega, Q, \Sigma, \delta, q_0, F)$ is an ordered triple $(q, w, r)$ where $q \in Q$ is the current state, $w \in \Sigma^*$ is the remaining input and $r \in C$ is the current bag.

We say that the configuration $(q_1, w_1, r_1)$ is contained in the configuration $(q_2, w_2, r_2)$ if $q_1 = q_2$, $w_1 = w_2$ and $r_1 \leq r_2$.

We will now define the consequence relations to be used with a bag automaton $M = (\Omega, Q, \Sigma, \delta, q_0, F)$.

For each $y \in \Gamma_0$, let $\vdash^y_M$ be the consequence relation (or simply $\vdash^y$ if there is no confusion) on $Q \times \Sigma^* \times C$ defined as follows:

For $a \in \Sigma \cup \{\lambda\}, (p, aw, r) \vdash^y (p', w, r')$ if there exists a transition from $\delta(p, a, y)$ to $(p', i)$, $r' = i(r)$ is defined and either $y \in \text{supp}(r)$ or $r = y = 0$.

Furthermore, we say $y$ is being examined by the bag.

Similar to taking the reflexive, transitive closure, we extend the consequence relation to words $x \in \Gamma_0^*$ by:

$$
\begin{align*}
(p, w, r) & \vdash^\lambda (p, w, r), \\
(p, w, r) & \vdash^x (p', w', r') \text{ if } (p, w, r) \vdash^y (p_1, w_1, r_1) \vdash^{x_1} (p', w', r'), \\
& \text{for some } p_1 \in Q, w_1 \in \Sigma^*, r_1 \in C \text{ with } x = yx_1, y \in \Gamma_0,
\end{align*}
$$

for all $p, p' \in Q, w, w' \in \Sigma^*, r, r' \in C$.

The language $L(M)$ accepted by $M = (\Omega, Q, \Sigma, \delta, q_0, F)$ is

$$L(M) = \{w \mid (q_0, w, 0) \vdash^x (q_f, \lambda, r), \text{ for some } q_f \in F, r \in C, x \in \Gamma_0^*\}.$$  

Let $M = (\Omega, Q, \Sigma, \delta, q_0, F)$ be a bag automaton. We say $M$ enters an infinite loop on $w$ if for all $j > 0$, there exists $\alpha_1 \in \Gamma_0^*$ such that $|\alpha_1| = j$ and

$$(q_0, w, 0) \vdash^{\alpha_1} (q_1, w_1, r_1),$$

for some $q_1, w_1, r_1$.

Let $M = (\Omega, Q, \Sigma, \delta, q_0, F)$ be a bag automaton. We say $M$ is well-formed if:

1. $$(q_0, w, 0) \vdash^{\alpha_1} (q_i, w_i, r_i) \vdash^y (q_{i+1}, w_{i+1}, r_{i+1}) \vdash^{\alpha_2} (q_f, \lambda, r_f),$$
   for some $q_f \in F, r_f \in C, \alpha_1, \alpha_2 \in \Gamma_0^*$, $y \in \Gamma_0$, then for every bag letter $x_j \in \text{supp}(r_i)$,

   $$(q_0, w, 0) \vdash^{\alpha_1} (q_i, w_i, r_i) \vdash^{x_j} (q_j, w_j, r_j) \vdash^{\alpha_2} (q_f', \lambda, r_f'),$$

   for some $q_f' \in F, r_f' \in C, \alpha_1' \in \Gamma_0^*$ and

2. $M$ does not enter an infinite loop on $w$.  

Intuitively, if a word is accepted using some sequence of bag letters, then it must get accepted if at any step, any other bag letter is examined instead. In this way, the bag can “choose any letter it wants” at any time to give the transition function. Furthermore, by enforcing that $M$ does not enter an infinite loop on any word that is accepted, we ensure that the automaton cannot wait an arbitrarily long time for the bag letter of its choosing. Indeed, this models closely the stochastic nature of solution. We note that Condition 1 is in fact useless (in terms of languages accepted) without Condition 2 for the reason discussed above.

One could also study the non-well-formed variant (bag automata without the well-formed condition). It is not difficult to show that these languages are in fact “almost” equal to the partially blind multicounter languages (see [5]). We say “almost” because bag automata are able to test for the empty bag. Thus, one can show that non-well-formed bag languages are equal to the smallest full AFL containing the partially blind languages. This can be used to show that they accept a language family properly between the partially blind and recursive languages. Indeed, the language $\{wcw^R \mid w \in \{a,b\}^*\}$ cannot be accepted by non-well-formed bag automata (and hence also well-formed bag automata). We will omit these results as they are not relevant to the stochastic nature we are trying to model.

The family of well-formed one-way, non-deterministic (and deterministic, respectively) bag automata is denoted by $\text{wnba}$ (and $\text{wdba}$, respectively).

The family of well-formed one-way, non-deterministic (and deterministic, respectively) bag languages is denoted by

$$\text{WNBL} = \{L(M) \mid M \in \text{wnba}\}.$$  \hspace{1cm} (6)

and

$$\text{WDBL} = \{L(M) \mid M \in \text{wdba}\}.$$  \hspace{1cm} (7)

Let $k \in \mathbb{N}_0$. We define $\text{wnba}^k = \{M = (\Omega, Q, \Sigma, \delta, q_0, F) \mid |\Gamma| \leq k\}$. Furthermore, we let

$$\text{WNBL}^k = \{L(M) \mid M \in \text{wnba}^k\}.$$  \hspace{1cm} (8)

Similarly for $\text{wdba}^k$.

**Example 1.** Consider the language

$$L = \{wca a^{\lfloor w \rfloor} \mid w \in \{a_1, a_2\}^+, a \in \text{alph}(w)\},$$  \hspace{1cm} (9)

(the projection of words onto letters). We construct a deterministic well-formed bag automaton $M = (\Omega, Q, \Sigma, \delta, q_0, F)$ accepting $L$. Let $\Omega = (I, C, I)$ be a bag with $I = \{a_1, a_2\}$, $Q = \{q_0, q_1, q_2, q_3, q_f\}$ is the set of states, $F = \{q_f\} \subseteq Q$ is the set of final states, $q_0 \in Q$ is the initial state and the transition function $\delta$ is defined as follows:
\[
\delta(q_0, b, a) = \{q_0, \text{toss}(b)\} \text{ for all } a \in \Gamma_0, b \in \{a_1, a_2\},
\]
\[
\delta(q_0, c, a) = \{q_1, \text{stay}\} \text{ for all } a \in \Gamma,
\]
\[
\delta(q_1, a_1, a) = \{q_2, \text{stay}\}, \quad \delta(q_1, a_2, a) = \{q_3, \text{stay}\} \text{ for all } a \in \Gamma,
\]
\[
\delta(q_2, a_1, a_1) = \{q_2, \text{grab}(a_1)\}, \quad \delta(q_2, a_2, a_1) = \{q_3, \text{grab}(a_1)\},
\]
\[
\delta(q_2, a_2, a_2) = \{q_3, \text{grab}(a_2)\}, \quad \delta(q_3, \lambda, a_1) = \{q_3, \text{grab}(a_1)\},
\]
\[
\delta(q_3, \lambda, a_2) = \{q_3, \text{grab}(a_2)\}, \quad \delta(q_3, \lambda, 0) = \{q_f, \text{stay}\}.
\]

Indeed, \(M\) tosses each letter of \(w\), one by one, into the bag. After \(M\) reads \(ca\), for some \(a\), it can sequentially discard every bag letter, however \(M\) only reads an input letter if the bag letter is equal to \(a\). It is clear that \(L(M) = L\) and it is also clear that \(M\) is well-formed since every word in \(L\) will accept regardless of the symbol examined by the bag at each step. If it is not the symbol that we want, we discard it and continue.

**Example 2.** Let \(L = \{a^n b^n c^n \mid n \geq 0\}\). We explain intuitively why \(L \notin \text{WNBL}\) (a formal proof is immediate from Lemma 2). As we read the \(a\)'s, we must toss both \(b\)'s and \(c\)'s into the bag at each step. However, when we read the \(b\)'s from the input tape, we cannot simply grab a \(b\) from the bag at each step, as \(c\) might be grabbed instead. Further, if we define the transition function to “stay” whenever \(c\) is grabbed from the bag, and thus wait for a \(b\), then the machine would enter an infinite loop on a word that is in the language (if it can happen once, it can happen arbitrarily many times). The only way to guarantee that the symbol \(b\) is grabbed is to discard all \(c\)'s until a \(b\) is obtained.

Before proceeding with discussion of well-formed bag languages in more detail we note that the following proposition can be easily shown.

**Proposition 1** Let \(k \in \mathbb{N}\). Let \(M = (\Omega, Q, \Sigma, \delta, q_0, F, Z_0) \in \text{nca}\) where \(\Gamma = \{X, Z_0\}\) such that the counter is increased by at most one per move (without loss of generality). Now construct a second machine \(M' = (\Omega, Q', \Sigma, \delta', q_0, F) \in \text{wdba}\) such that \(M'\) does not have any stay transitions and \(L(M) = L(M')\). Similarly for \(\text{wnba}\).

## 4 Well-Formed Bag Languages

We now consider the families of languages defined by well-formed bag automata. We begin by showing that the languages accepted by bag automata with non-deterministic control and a single letter bag alphabet are equivalent to the non-deterministic one-counter languages; a similar result holds for deterministic bag automata and deterministic one-counter languages.

**Proposition 2** \(\text{NCL} = \text{WNBL}^1\) and \(\text{DCL} = \text{WDBL}^1\).

**Proof.** Let \(M = (\Gamma, Q, \Sigma, \delta, q_0, F, Z_0) \in \text{nca}\) where \(\Gamma = \{X, Z_0\}\) such that the counter is increased by at most one per move (without loss of generality). Now construct a second machine \(M' \in \text{wnba}\) such that \(M' = (\Omega, Q, \Sigma, \delta', q_0, F)\) where \(\Omega = (\Gamma, C, I)\) and \(\Gamma' = \{X\}\). The mapping \(\delta'\) is constructed in the following way:

\[(p, i) \in \delta'(q, a, x) \text{ if and only if } (p, \gamma) \in \delta(q, a, y), \quad (10)\]
where \( x = y \) if \( y \neq Z_0 \) and \( x = 0 \) otherwise, \( i = \text{grab}(x) \) if \( \gamma = \lambda \), \( i = \text{stay} \) if \( \gamma = y \) and \( i = \text{toss}(X) \) otherwise, for any \( p, q \in Q, a \in \Sigma \cup \{ \lambda \} \).

Clearly, for \( w \in \Sigma^* \), \((q_0, w, Z_0) \vdash_M^* (q_f, \lambda, \gamma)\) for some \( q_f \in F, \gamma \in \Gamma \) if and only if there exists \( \alpha \in \Gamma_0^* \) such that \((q_0, w, 0) \vdash_M^* (q_f, \lambda, r)\) for some \( r \in C \).

In addition, we can assume that \( M \) does not enter an infinite loop since it is known that every one-counter language can be accepted by a one-counter automaton which applies at most \( k \) consecutive lambda transitions, for some \( k \in \mathbb{N}_0 \) (see [4]). Thus, we can assume \( M' \) is well-formed. The converse can be proven similarly.

It is also known that we can transform every deterministic one-counter automaton into another which does not go into an infinite loop and accepts the same language [8], thus the proposition follows. \( \square \)

Also, as noted in the introduction, it can be shown (in fact even without the well-formed condition) that:

**Lemma 1** \( \{wcw^R \mid w \in \{0,1\}^*\} \notin \text{WNBL, WDBL}. \)

We omit this proof due to space considerations.

Although the family of partially blind multicounter languages (and hence bag automata without the well-formed restriction) contains non context-free languages, for example \( \{a^n b^n c^n \mid n \geq 0\} \), if the well-formed condition holds, the resulting language family is inside the context-free languages (see Example 2).

**Lemma 2** \( \text{WNBL} \subseteq \text{CF and WDBL} \subseteq \text{DCF}. \)

**Proof.** Let \( M = (Q, \Sigma, \delta, q_0, F) \in \text{wnba} \). We construct a pushdown automaton \( M' = (\Gamma', Q, \Sigma, \delta', q_0, F, Z_0) \in \text{pda} \) such that \( \Gamma' = \Gamma \cup \{ Z_0 \} \). The mapping \( \delta' \) is constructed in the following way:

\[
(p, i) \in \delta(q, a, y) \text{ if and only if } (p, \gamma) \in \delta'(q, a, x),
\]

where \( x = y \) if \( y \neq 0 \) and \( x = Z_0 \) otherwise, \( \gamma = \lambda \) if \( i = \text{grab}(y) \), \( \gamma = x \) if \( i = \text{stay} \) and \( \gamma = x z \) if \( i = \text{toss}(z) \), for any \( p, q \in Q, a \in \Sigma \cup \{ \lambda \} \).

Now, assume by induction, that \((q_0, w, 0) \vdash_M^* (q_1, w_1, r_1) \vdash_M^* (q_f, \lambda, r_f)\) and that \((q_0, w, Z_0) \vdash_M^* (q_1, w_1, \gamma)\) where \( \alpha_1, \alpha_2 \in \Gamma_0^* \), \( w_1 \in \Sigma^* \), \( q_1 \in Q \), \( q_f \in F \) and \( r_1(z) = |\gamma_1|_z, \forall z \in \Gamma \). Let \( x \) be the rightmost symbol of \( \gamma_1 \) (top of the pushdown). Then, since \( M \) is well-formed, \((q_1, w_1, r_1) \vdash_M^* (q'_2, w'_2, r'_2) \vdash_M^* (q_f', \lambda, r_f')\) and thus we find \((q_1, w_1, \gamma) \vdash_M^* (q'_2, w'_2, \gamma'_2)\) where \( |\gamma'_2|_z = r'_2(z), \forall z \in \Gamma \). However, we note that it is possible that when a different letter is picked from the bag, the size of the remaining derivation could increase (ie. \( |\alpha'_2| \) could be bigger than \( |\alpha_2| - 1 \) above). Although this may happen, it cannot happen indefinitely as \( M \) cannot enter an infinite loop. Hence, by induction, \( w \in L(M') \).

Let \( w \notin L(M) \). Then no possible sequence of symbols examined by the bag will result in a successful computation. In particular, if the symbol examined by the bag is the symbol at the top of the pushdown, there is not a sequence of transitions of \( M' \) accepting \( w \). Thus, \( w \notin L(M') \).
Hence, \( L(M) = L(M') \).
Similarly for the deterministic case.
Both inclusions can be seen to be strict by Lemma 1. \( \Box \)

Although it is decidable whether a non-deterministic bag automaton with one bag letter that does not enter an infinite loop is well-formed, if we allow even one occurrence of a second letter, the following proposition indicates that this is no longer the case.

**Proposition 3** Let \( M \) be a non-deterministic bag automaton (not necessarily well-formed). The question “Is \( M \) well-formed?” is undecidable.

**Proof.** It is known that equality is undecidable for \( NCL \) (see [1]). Let \( M_1, M_2 \in nca \) such that it is undecidable whether \( L(M_1) = L(M_2) \) (we can assume that these make a finite number of consecutive \( \lambda \) transitions). It is easy to construct \( M_3, M_4 \in \text{wnba}^1 \) as in Proposition 2 such that \( L(M_1) = L(M_3), L(M_2) = L(M_4) \) and \( M_3 \) and \( M_4 \) have disjoint sets of states.

We construct \( M \in \text{nsa}^2 \) such that at the beginning of the computation, we toss new letters \( x \) and \( y \) into the bag. We then grab one. If it is an \( x \), then discard the other and do \( M_3 \). If it is a \( y \), then discard the other and do \( M_4 \). Then, \( M \) is well-formed if and only if \( L(M_3) = L(M_4) \) and \( M \) is undecidable.

Hence, it is undecidable whether a non-deterministic bag automaton is well-formed. \( \Box \)

We wish to show that the hierarchy of deterministic, well-formed bag automata with respect to the number of bag characters is strict. We first need the following two lemmata. The following construction keeps a finite pushdown in the second coordinate of the states and often uses the pushdown instead of the bag when applying lambda transitions as indicated. Let \( k \geq 1 \).

**Lemma 3** Let \( M = (\Omega, Q, \Sigma, \delta, q_0, F) \in \text{wnba}^k \) without stay transitions. Then we can construct a bag automaton \( M' = (\Omega, Q', \Sigma, \delta', q'_0, F') \in \text{wnba}^k \) such that during any computation of a word accepted by \( M \), there are at most \( |Q| \) toss moves between reading any two input letters and they occur consecutively directly after reading the first input letter. Furthermore, \( M \) is deterministic if and only if \( M' \) is deterministic and \( L(M) = L(M') \).

**Proof.** We let \( n = |Q| \). Let \( \bar{Q} = \{ \bar{q} \mid q \in Q \} \), \( Q' = (Q \cup \bar{Q}) \times I^n_0 \), \( q'_0 = [q_0, 00 \ldots 0] \) and \( F' = F \times I^n_0 \). That is, we keep a pushdown of size \( n \) in the state, which we always pad with 0 as the blank symbol.

If \((q_1, \text{toss}(y)) \in \delta(q, \lambda, x)\), then change to
\[
([q_1, axy00 \ldots 0], \text{stay}) \in \delta'([q, ax00 \ldots 0], \lambda, z),
\]
\[
([q_1, y00 \ldots 0], \text{stay}) \in \delta'([q, 00 \ldots 0], \lambda, x), \forall z \in I^n_0, \alpha \in \bigcup_{j=1}^{n-2} I^j.
\]

If \((q_1, \text{grab}(x)) \in \delta(q, \lambda, x)\), then change to
\[
([q_1, 00 \ldots 0], \text{grab}(x)) \in \delta'([q, 00 \ldots 0], \lambda, x),
\]
\[
([q_1, ax00 \ldots 0], \text{stay}) \in \delta'([q, ax00 \ldots 0], \lambda, z), \forall z \in I^n_0, \alpha \in \bigcup_{j=1}^{n-1} I^j.
\]
Bag Automata and Stochastic Retrieval of Biomolecules in Solution

247

If (q1 , i) ∈ δ(q, a, x), a ∈ Σ, x ∈ Γ0 , i ∈ I then change to
([q1 , 00 · · · 0], i) ∈ δ  ([q, 00 · · · 0], a, x),
([q¯1 , αy00 · · · 0], stay) ∈ δ  ([q, αx00 · · · 0], a, z), where y = λ,
if i = grab(x), y = x if i = stay and y = xu if i = toss(u),

j
∀z ∈ Γ0 , α ∈ n−1
j=1 Γ , unless |αy| = n + 1. Also, add
([q¯1 , xi1 · · · xij−1 00 · · · 0], toss(xij )) ∈ δ  ([q¯1 , xi1 · · · xij 00 · · · 0], z),
∀z ∈ Γ0 , xil ∈ Γ, 1 < j ≤ n,
([q1 , 00 · · · 0], toss(xi1 )) ∈ δ  ([q¯1 , xi1 00 · · · 0], z), ∀z ∈ Γ, xi1 ∈ Γ .
It is clear that M  is deterministic if and only if M is.
“⊆”. Let w ∈ L(M ). Notice that every possible sequence of bag symbols
examined by the bag must lead to an accepting computation. In particular, if
the symbol at the top of a pushdown is always the symbol examined by the
bag, then the computation must lead to acceptance (similar to the proof of
WNBL ⊆ CF). Consider such an accepting computation of w. The states of M 
keep track of a pushdown of length n (with 0 being used as a blank symbol).
Whenever a lambda transition is applied in which we toss y we push y onto the
ﬁnite pushdown instead (we will indicate shortly why the pushdown of size n
suﬃces). Whenever a lambda transition is applied in which we grab x, if there
are any symbols in the ﬁnite pushdown, the top symbol is x, which we pop, then
continue. If the pushdown is empty, then we perform the transition as in M using
the bag. Whenever an input letter is read, if there are any letters on the ﬁnite
pushdown, we toss them one by one into the bag using lambda transitions, before
continuing the computation with the ﬁnite pushdown empty. These are the only
toss instructions that occur on lambda using M  . Thus, assuming we never need
more than n symbols on the pushdown, then L(M ) ⊆ L(M  ). We know that
during the computation of an accepting word w, M cannot toss more than n
symbols consecutively using lambda transitions otherwise M would enter an
inﬁnite loop on w (since then M  would enter the same state twice with the ﬁrst
conﬁguration contained in the second). Furthermore, each consecutive sequence
of toss instructions on lambda must each occur using diﬀerent states. Thus, the
only situation in which we would need to keep track of more than n symbols
on the state would be if 0 < l1 ≤ n symbols were pushed onto the pushdown,
for some l1 , then p1 < l1 symbols were popped, then more symbols were pushed
eventually raising the pushdown above n symbols (we need not consider the
case where l1 ≥ p1 since the pushdown would then be empty). Assume that
at least n + 1 symbols are required on the pushdown. Then, when each of the
n + 1 symbols were pushed, M must have been in a diﬀerent state for each,
otherwise, qi was entered twice, say, and the bag in M must have had the ﬁrst
conﬁguration contained in the second, thus entering an inﬁnite loop.
“⊇” similar to “⊇” of WNBL ⊆ CF of Lemma 2.
Moreover, since M is well-formed and all other transitions added are for all
possible bag letters, M  is also well-formed. 
The next lemma states that at any step of any computation of a word that is
accepted by a well-formed automaton, the number of steps in the computation
so far is linear in the length of the input word read so far.


Lemma 4 Let $M = (Q, \Sigma, \delta, q_0, F) \in \text{wuba}$ such that $M$ satisfies the conditions of Lemma 3. Then there exists $l \geq 0$ such that
\[
(q_0, w, 0) \vdash_{\alpha_k'} (q'_{w_k}, w'_k) \vdash x (q_{w_k}, w_k, r_{w_k}) \vdash_{\beta} (q_f, \lambda, r),
\]
for some $x \in \Gamma_0, w'_k \neq w_k, r \in C$, $\alpha_k', \beta \in \Gamma_0^*$, $q_f \in F$ and we let $\alpha_k = \alpha_k' x$, then
\[
|\alpha_k| \leq l(|w| - |w_k|).
\]
In particular, $l$ can be any integer greater than $|Q|^2$.

Proof. Let $n = |Q|$. Every toss or grab instruction increases the size of the derivation by one. Each toss instruction increases the size of the bag by one. Furthermore, toss instructions can only occur consecutively immediately after reading a letter as input or while reading a letter. The construction of Lemma 3 introduces many stay transitions, so we must account for these. There are at most $n - 2$ stay transitions between each grab, otherwise $M$ would enter an infinite loop (there must be at least one toss and one grab transition, so $n - 2$ is enough), and there are no stay transitions during the consecutive toss transitions. The size of $r_{w_k}$ is at most $n(|w| - |w_k|)$. For every element tossed into the bag, there are at most $n$ transitions applied (one toss, one grab, $n - 2$ stays). Hence, $|\alpha_k| \leq n^2(|w| - |w_k|)$. □

Finally, we will show using the two previous lemmata that the deterministic, well-formed hierarchy with respect to number of bag symbols is strict.

Let $k \geq 1$. We use the following languages to separate the hierarchy:
\[
L_k = \{wca_i^{\lambda a_i^+} | \ w \in \{a_1, \ldots, a_k\}^+, a_i \in \{a_1, \ldots, a_k\}\}.
\]

Lemma 5 Let $L_{k+1} \in \text{WDBL}^{k+1} - \text{WDBL}^k$.

Proof. It is easily seen that $L_{k+1} \in \text{WDBL}^{k+1}$. By contradiction, assume $L_{k+1} \in \text{WDBL}^k$. Thus, there exists $M = (Q, \Sigma, \delta, q_0, F) \in \text{wdba}^k$ which satisfies the conditions of Lemma 4, $\Gamma = \{x_1, \ldots, x_k\}, \Sigma = \{a_1, \ldots, a_k, a_{k+1}, c\}$ and $L(M) = L_{k+1}$. Hence,
\[
(q_0, wa_i^{\lambda a_i^+}, 0) \vdash_{\alpha_i'} (q'_w, a_i ca_i^{\lambda a_i^+}, r'_w) \vdash x (q_w, ca_i^{\lambda a_i^+}, r_w) \vdash_{\beta} (q_{f_i}, \lambda, r),
\]
for some $q_{f_i} \in F, \alpha_i', \beta \in \Gamma_0^*, x \in \Gamma_0, a_i \in \Sigma, r \in C$ and we let $\alpha_1 = \alpha_1' x$.

Assume $\text{perm}(w_1) \neq \text{perm}(w_2), w_{a_1} = w_{a_2} = w$ and $r_{w_1} = r_{w_2}$. Then there is a $b \in \Sigma$ such that $|w_1|_b \neq |w_2|_b$. However, since $M$ is well-formed and deterministic, every path of the automaton must accept. Thus, $w_1b^{\lambda a_1|b}$ and $w_2b^{\lambda a_2|b}$ will both accept, a contradiction.

Hence, for every two words $w_1, w_2$ such that $\text{perm}(w_1) \neq \text{perm}(w_2)$, we have that $(q_{w_1}, ca_i^{\lambda a_i|b}, r_{w_1}) \neq (q_{w_2}, ca_i^{\lambda a_i|b}, r_{w_2})$.

By Lemma 4, there exists $l \geq 0$ such that $|\alpha_1| \leq l|w|$. Let $n = |Q|$ and let $s_0 \in \mathbb{N}$ such that $s_0^{k+1} > n((l+1)(k+1)s_0 + l)^k + n$ which exists for all fixed $k, n, l$. Let
Bag Automata and Stochastic Retrieval of Biomolecules in Solution

$$U(s_0) = \{ w \mid a \in \text{alph}(w), |w|_a \leq s_0, \forall a \}. \text{ For each } w \in U(s_0), wca|w|_a \in L(M) \text{ for each } a. \text{ Hence, there exists } q_w, r_w \text{ such that}$$

$$(q_0, wca|w|_a, 0) \vdash_{\beta_1} (q_w, ca|w|_a, r_w),$$

for each $$a$$. So, there are $$s_0^{k+1}$$ words in $$U(s_0)$$ that have different permutation closures and so $$s_0^{k+1}$$ distinct intermediate configurations.

Let $$w \in U(s_0)$$. Then $$(q_0, w, 0) \vdash_\alpha (q_w, \lambda, r_w)$$. Notice $$|w| \leq |k+1|s_0$$. Since $$|\alpha| \leq l|w|$$, it follows that $$r_w(x) \leq (l+1)(k+1)s_0 + l$$ for each $$x \in I$$. Moreover, when $$x = 0$$, then $$r_w = 0$$. Thus, there are at most $$((l+1)(k+1)s_0 + l)^k + 1$$ different $$r_w$$ and thus at most $$n((l+1)(k+1)s_0 + l)^k + n$$ different pairs $$q_2, r_w$$, a contradiction.

Hence, $$L_{k+1} \in \text{WDBL}^{k+1} - \text{WDBL}^k \square$$

**Theorem 1** Let $$k \in \mathbb{N}$$. Then

$$\text{DCL} = \text{WDBL}^1 \subset \text{WDBL}^2 \subset \cdots \subset \text{WDBL}^k \subset \cdots \subset \text{WDBL} \subset \text{DCF}.$$  

5 Conclusions

The family of languages accepted by deterministic well-formed bag automata was shown to be strictly between the family of deterministic counter languages and the family of deterministic context-free languages. Similarly, we show that the languages accepted by the nondeterministic version are between the nondeterministic counter and context-free languages. Further, it was shown that the deterministic well-formed bag languages form a strict hierarchy. It is an open problem whether the non-deterministic well-formed languages also form a strict hierarchy. We conjecture that it collapses and equals the non-deterministic counter languages. This would be interesting theoretically as then the deterministic well-formed hierarchy would be between the deterministic and non-deterministic counter languages, an unusual place for a strict, infinite hierarchy.

The well-formed model relates to the study of non-addressable memory in biocomputing, with a particular interest in well-formed bag automata which model non-addressable storage in a practical fashion. It is hoped that further investigation and results will give us insights into not only the theoretical realm of non-deterministic storage, but also allow us to precisely define the limits of stochastic retrieval in practical biocomputing.

References


An Optimal Algorithm
for Maximum-Sum Segment and Its Application
in Bioinformatics

Extended Abstract

Tsai-Hung Fan\textsuperscript{1}, Shufen Lee\textsuperscript{2}, Hsueh-I Lu \textsuperscript{*}\textsuperscript{3}, Tsung-Shan Tsou\textsuperscript{1},
Tsai-Cheng Wang\textsuperscript{2}, and Adam Yao\textsuperscript{2}

\textsuperscript{1} Institute of Statistics
National Central University
Chung-li 320, Taiwan, R.O.C.
\textsuperscript{2} Institute of Biomedical Sciences
Academia Sinica
Taipei 115, Taiwan, R.O.C.
\textsuperscript{3} Institute of Information Science
Academia Sinica
Taipei 115, Taiwan, R.O.C.

hil@iis.sinica.edu.tw
www.iis.sinica.edu.tw/~hil/

\textbf{Abstract.} We study a fundamental sequence algorithm arising from bioinformatics. Given two integers $L$ and $U$ and a sequence $A$ of $n$ numbers, the maximum-sum segment problem is to find a segment $A[i,j]$ of $A$ with $L \leq j - i + 1 \leq U$ that maximizes $A[i] + A[i+1] + \cdots + A[j]$. The problem finds applications in finding repeats, designing low complexity filter, and locating segments with rich C+G content for biomolecular sequences. The best known algorithm, due to Lin, Jiang, and Chao, runs in $O(n)$ time, based upon a clever technique called left-negative decomposition for $A$. In the present paper, we present a new $O(n)$-time algorithm that bypasses the left-negative decomposition. As a result, our algorithm has the capability to handle the input sequence in an online manner, which is clearly an important feature to cope with genome-scale sequences. We also show how to exploit the sparsity in the input sequence: If $A$ is representable in $O(k)$ space in some format, then our algorithm runs in $O(k)$ time. Moreover, practical implementation of our algorithm running on the rice genome helps us to identify a very long repeat structure in rice chromosome 1 that is previously unknown.

* Corresponding author. Research of this author is supported in part by NSC Grant 91-2215-E-011-001.
1 Introduction

Let $A$ be an input sequence of $n$ numbers, where $A[i]$ denotes the $i$-th number in $A$. For any indices $i$ and $j$ with $1 \leq i \leq j \leq n$, let

$$w(i, j) = j - i + 1;$$

Given two integers $L$ and $U$ with $1 \leq L \leq U \leq n$, the maximum-sum segment problem is to find indices $i$ and $j$ with $L \leq w(i, j) \leq U$ that maximizes $s(i, j)$. The problem finds useful applications in bioinformatics including (a) finding tandem repeats, which are commonly used to mapping disease genes \cite{7}, (b) locating DNA segments with rich C+G content, which is a key step in gene finding and promoter prediction \cite{2, 3}, and (c) low complexity filter, which is mostly applied in sequence database search \cite{1}. It is not difficult to see that the maximum-sum segment problem can be solved in $O(n(U - L + 1))$ time. Lin, Jiang, and Chao \cite{4} gave the first known $O(n)$-time algorithm for this problem, based upon a clever but complicated technique called the left-negative decomposition of the input sequence $A$. In this paper, we give a new $O(n)$-time algorithm that bypasses the preprocessing step for left-negative decomposition and is conceptually simpler.

Our algorithm is good for practical implementation. Firstly, our algorithm easily has the capability to handle the input sequence in an online manner, i.e., does not require any preprocessing on the input sequence, and “interactively” outputs the starting and ending indices of the best segment so far right after it obtains the $j$-th number $A[j]$ for each index $j \geq L$. Running in an online manner is clearly an important feature for coping with genome-scale DNA sequences \cite{6}, since otherwise the amount of required main memory would make any algorithm impractical. Secondly, for bioinformatics application like finding repeats, the input sequence is usually quite sparse. In the paper, we explain how to exploit the sparsity of the input sequence. Practical experiments of our methods in finding repeats are interesting: Rice chromosome 1 (43Mbp) masked by our previously developed repeat-mask program and simplicity filter program was pairwisely aligned against itself. The aligned fragments (e-value $< e^{-10}$) are sparsely distributed over chromosome 1. By applying our algorithm on this sparse sequence, we found a new repetitive fragment as shown in Figure 1, which has over 80 homologues (e-value $< e^{-10}$) in rice chromosome 1 but none in the latest TIGR Rice Repeat Database released on July 29, 2002 \cite{5}.
The rest of the paper is organized as follows. Section 2 gives our linear-time algorithm. Section 3 shows how to implement our algorithm to process the input sequence in an on-line manner. Section 4 shows how our algorithm exploits the sparsity of the input sequence.

2 Our Algorithm

For notational brevity, let $s(i, j) = 0$ for any indices $i$ and $j$ with $i > j$. Also, let $s(j) = s(1, j)$ for each index $j$. For each index $j = 1, 2, \ldots, n$, let $b(j)$ be the largest index $i$ with $L \leq w(i, j) \leq U$ that minimizes $s(1, i - 1)$. By $s(i, j) = s(1, j) - s(1, i - 1)$, one can easily see that $b(j)$ is an index $i$ with $L \leq w(i, j) \leq U$ that maximizes $s(i, j)$. Therefore, for the purpose of finding an optimal pair of indices $i$ and $j$ in $O(n)$ time, it suffices to show how to compute $b(1), b(2), \ldots, b(n)$ in $O(n)$ time.

For each index $j = 1, 2, \ldots, n$, let

$$\ell_j = \max(j - U + 1, 1);$$

$$r_j = j - L + 1.$$  \hspace{1cm} (1)

In other words, $\ell_j$ is the smallest index $i$ with $w(i, j) \leq U$, and $r_j$ is the largest index $i$ with $w(i, j) \geq L$. (See Figure 2.) If $U = n$ (i.e., the length upper bound $U$ is ineffective), then one can easily come up with an $O(n)$-time algorithm by observing that $b(j + 1)$ is either $b(j)$ or $r_j + 1$. More precisely, we have $b(L) = 1$ and

$$b(j + 1) = \begin{cases} b(j) & \text{if } s(r_j - 1) > s(b(j) - 1); \\ r_{j+1} & \text{otherwise} \end{cases}$$

for each index $j$ with $L \leq j < n$.

To cope with the case with effective length upper bound $U$, we need the following data structure, which can be appropriately maintained in overall $O(n)$ time throughout the execution of our algorithm. For each index $j = L, \ldots, n$, let $B_j$ be the sequence of $t_j$ monotonically increasing indices such that $B_j[1] = b(j)$, $B_j[t_j] = r_j$, and $B_j[k]$ is the largest index $i$ with $B_j[k - 1] < i \leq r_j$ that minimizes $s(i - 1)$ for each $k = 2, 3, \ldots, t_j$. One can verify that $B_{j+1}$ can be obtained from $B_j$ as follows.

---

atataataatattagagatatagaggtcatataaatatatattaqatatataatag
aatattcggtattagataaataacagatatagagatagattcattatatagatactatatagagataatagatatatatatagctata
Fig. 1. A repeat in rice chromosome 1 identified by our algorithm
Let $B_j[t_j + 1] = r_j$. Let $t'_j$ be the smallest index $k$ with $k \leq t_j + 1$ such that $s(B_j[k] - 1) \geq s(r_j - 1)$. If $b(j) \geq \ell_j + 1$, then let $\delta_j = 0$; otherwise, let $\delta_j = 1$. Then we have $t_{j+1} = t'_j - \delta_j$ and $B_{j+1}[k] = B_j[k + \delta_j]$ for each $k = 1, 2, \ldots, t_{j+1}$.

Therefore, our algorithm for computing all $b(j)$ is as shown in Figure 4. We use an array $B$ with two indices $x$ and $y$ to implement the aforementioned arrays $B_j$. More specifically, in the iteration for index $j$, the entries of $B_j$ are recorded as $B[x], B[x+1], \ldots, B[y]$. Figure 3 is an illustration for $B$, where $\phi(p, q)$ denotes the largest index $i$ with $p \leq i \leq q$ and $s(i) = \min \{s(p), s(p+1), \ldots, s(q)\}$. Whenever $\delta_j = 1$, we increment the value of $x$ will be incremented. To see that the algorithm indeed runs in $O(n)$ time, one can verify that (1) $y - x = 0$ holds initially and $y \geq x$ holds throughout the execution, (2) the value of $y - x$ is increased by one for $O(n)$ times, and (3) each iteration of the while-loop decreases the value of $y - x$ by one.

### 3 The Online Version

As a matter of fact, the algorithm shown in Figure 4 is already very close to be one that has the capability of processing the input sequence in an
An Optimal Algorithm for Maximum-Sum Segment and Its Application

algorithm maxsum {
  let $B[1] = b(L) = x = y = 1$;
  for $j = L + 1$ to $n$ do {
    while $x \leq y$ and $s(B[y] - 1) \geq s(r_j - 1)$ do
      let $y = y - 1$;
      let $y = y + 1$;
      let $B[y] = r_j$;
      if $B[x] < \ell_j$ then
        let $x = x + 1$;
    output $b(j) = B[x]$;
  }
}

Fig. 4. Our $O(n)$-time algorithm for computing all $b(j)$

on-line manner. Since each $\ell_j$ and $r_j$ can be computed in $O(1)$ time as shown in Equation (1), it suffices to show how to take care of those values $s(\cdot)$ used in the algorithm. Fortunately, each of them has index less than or equal to $j$. Therefore, it is not difficult to modify the algorithm in Figure 4 into the on-line version shown in Figure 5, where we use $s[i]$ to keep the values of $s(i)$ that have been computed on the fly so far.

algorithm online_maxsum {
  for $j = 2$ to $L$ do
    let $B[1] = x = y = 1$;
    output $b(L) = 1$;
  for $j = L + 1$ to $n$ do {
    while $x \leq y$ and $s[B[y] - 1] \geq s[r_j - 1]$ do
      let $y = y - 1$;
      let $y = y + 1$;
      let $B[y] = r_j$;
    if $B[x] < \ell_j$ then
      let $x = x + 1$;
    output $b(j) = B[x]$;
  }
}

Fig. 5. The on-line version of our algorithm
4 Exploiting Sparsity

We now consider the situation that all but $p$ numbers of $A$ are $c$ and all those non-$c$ entries are larger than $c$. That is the input sequence $A$ can be represented by (a) the “background” number $c$ and (2) $p$ index-value pairs $(j_1,A[j_1]),\ldots,(j_p,A[j_p])$, where $A[j_i] > c$ holds for each $i = 1,2,\ldots,p$. For instance, one may set $c \leq 0$ and $A[j_i] > 0$ for each $i = 1,2,\ldots,p$. As a matter of fact if $c \geq 0$, the lower bound $L$ on the segment length becomes ineffective, and thus the problem becomes trivial. Without loss of generality, let us assume $j_1 < j_2 < \cdots < j_p$. We show how to solve the problem in $O(p)$ time.

Clearly, it suffices to look for segments whose ending indices are among $j_1,j_2,\ldots,j_p$. Therefore, the problem can be reduced to computing $b(j_i)$ for each $i = 1,2,\ldots,p$. It is clear from the settings of our input that $B[j][i]$ with $1 \leq i \leq t_j$ has to be one of $j_1,j_2,\ldots,j_p$. (To see this, one can verify that each index $k = B[j][i]$ for $1 \leq i < t_j$ has to satisfy $A[k-1] \leq 0$ and $A[k] > 0$.) Observe that our algorithm, as shown in Figure 4, needs only the values of $s(B[j][i] - 1)$, which we can easily precompute in $O(p)$ time. Therefore, an $O(p)$-time algorithm for the maximum-sum segment problem is as shown in Figure 6.

```
algorithm sparse_maxsum {
    let $s[j_1-1] = c \cdot (j_1-1)$;
    for $i = 2$ to $p$ do {
        let $s[j_i-1] = s[j_i-1-1] + A[j_i-1] + c \cdot (j_i-j_i-1-1)$;
        let $B[1] = x = y = 1$;
        let $i_0$ be the smallest index $i$ with $j_i \geq L$;
        for $i = i_0$ to $p$ do {
            while $x \leq y$ and $s[B[y]-1] \geq s[r_{j_i}-1]$ do {
                let $y = y - 1$;
                let $y = y + 1$;
                let $B[y] = r_{j_i}$;
                if $B[x] < \ell_{j_i}$ then {
                    let $x = x + 1$;
                }
                output $b(j_i) = B[x]$;
            }
        }
    }
}
```

Fig. 6. Our $O(p)$-time algorithm for handling input representable in $O(p)$ space.
Acknowledgment

We thank the anonymous reviewers for their helpful comments.

References


Robust Parsing Using Dynamic Programming

Manuel Vilares¹, Victor M. Darriba¹, Jesús Vilares², and Leandro Rodríguez¹

¹ Department of Computer Science
University of Vigo
Campus As Lagoas s/n, 32004 Orense, Spain
{vilares,darriba,leandro}@uvigo.es

² Department of Computer Science
University of A Coruña
Campus de Elviña s/n, 15071 A Coruña, Spain
jvilares@udc.es

Abstract. A robust parser for context-free grammars, based on a dynamic programming architecture, is described. We integrate a regional error repair algorithm and a strategy to deal with incomplete sentences including unknown parts of unknown length. Experimental tests prove the validity of the approach, illustrating the perspectives for its application in real systems over a variety of different situations, as well as the causes underlying the computational behavior observed.

1 Introduction

An ongoing question in the design of parsers is how to gain efficiency in dealing with unexpected input, and to do so without either over-generating or under-generating. This supposes the capacity to deal with gaps, incorrectness and noise contained in the input, which are often the consequence of external deterioration due to transcription errors and human performance deviations, typically in natural language processing, speech recognition or even traditional programming tasks. At this point, robustness should be conceived as the ability to handle non-standard input, and to interpret it in order to generate a plausible interpretation.

Our goal is syntactic, and no attention is devoted to ill-formed lexicons or semantic correction, focusing instead on enhancing robustness with respect to parse errors, incompleteness or deviations from standard language. To comply with these requests, we integrate an error repair algorithm [10] and a strategy to deal with incomplete sentences, in a parser for context-free grammars (CFG’s). In this sense, we provide a dynamic programming approach which allows us both to save in computational efficiency and to simplify the formal definition framework.

* Research partially supported by the Spanish Government under projects TIC2000-0370-C02-01 and HP2002-0081, and the Autonomous Government of Galicia under projects PGIDIT01PXI10506PN, PGIDIT02PXIB30501PR and PGIDIT02SIN01E.
2 The Standard Parser

Our aim is to parse a sentence $w_1\ldots w_n = w_1\ldots w_n$ according to an unrestricted CFG $G = (N, \Sigma, P, S)$, where $N$ is the set of non-terminals, $\Sigma$ the set of terminal symbols, $P$ the rules and $S$ the start symbol. The empty string will be represented by $\varepsilon$. We generate from $G$ a push-down automaton (PDA) for the language $L(G)$. In practice, we choose an LALR(1) [1] device generated by ICE$^T$ [9], although any shift-reduce strategy is adequate. A PDA is a 7-tuple $A = (Q, \Sigma, \Delta, \delta, q_0, Z_0, Q_f)$ where: $Q$ is the set of states, $\Sigma$ the set of input symbols, $\Delta$ the set of stack symbols, $q_0$ the initial state, $Z_0$ the initial stack symbol, $Q_f$ the set of final states, and $\delta$ a finite set of transitions of the form $\delta(p, X, a) \ni (q, Y)$ with $p, q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ and $X, Y \in \Delta \cup \{\varepsilon\}$. Let the PDA be in a configuration $(p, X\alpha, ax)$, where $p$ is the current state, $X\alpha$ is the stack contents with $X$ on the top, and $ax$ is the remaining input where the symbol $a$ is the next to be shifted, $x \in \Sigma^*$. The application of $\delta(p, X, a)$ results in a new configuration $(q, Y\alpha, x)$ where $a$ has been scanned, $X$ has been popped, and $Y$ has been pushed.

To get polynomial complexity, we avoid duplicating stack contents when ambiguity arises. Instead of storing all the information about a configuration, we determine the information we need to trace in order to retrieve it. This information is stored in a table $I$ of items, $I = \{[q, X, i, j], q \in Q, X \in \{\varepsilon\} \cup \{\nabla_{r,s}\}, 0 \leq i \leq j\}$; where $q$ is the current state, $X$ is the top of the stack, and the positions $i$ and $j$ indicate the substring $w_i+1w_j$ spanned by the last terminal shifted to the stack or by the last production reduced. The symbol $\nabla_{r,s}$ indicates that the part $A_{r,s+1}\ldots A_{r,n_r}$ of a rule $A_{r,0} \rightarrow A_{r,1}\ldots A_{r,n_r}$ has been recognized.

We describe the parser using parsing schemata [7]; a triple $\langle I, \mathcal{H}, \mathcal{D} \rangle$, with $I$ the table of items previously defined, $\mathcal{H} = \{[a, i, i+1], a = w_{i+1}\}$ an initial set of triples called hypotheses that encodes the sentence to be parsed$^2$, and $\mathcal{D}$ a set of deduction steps that allow new items to be derived from already known ones. Deduction steps are of the form $\{\eta_1, \ldots, \eta_k \vdash \xi / \text{conds}\}$, meaning that if all antecedents $\eta_i \in I$ are present and the conditions $\text{conds}$ are satisfied, then the consequent $\xi \in I$ should be generated. In the case of ICE, $\mathcal{D} = \mathcal{D}_{\text{init}} \cup \mathcal{D}_{\text{shift}} \cup \mathcal{D}_{\text{sel}} \cup \mathcal{D}_{\text{red}} \cup \mathcal{D}_{\text{head}}$, where:

$$
\begin{align*}
\mathcal{D}_{\text{shift}} & = \{[q, \varepsilon, i, j] \vdash [q', \varepsilon, j, j+1] \mid \exists [a, j, j+1] \in \mathcal{H} \text{ shift}_{q'} \in \text{action}(q, a)\} \\
\mathcal{D}_{\text{sel}} & = \{[q, \varepsilon, i, j] \vdash [q, \nabla_{r,n_r}, i, j] \mid \exists [a, j, j+1] \in \mathcal{H} \text{ reduce}_{r,s} \in \text{action}(q, a)\} \\
\mathcal{D}_{\text{red}} & = \{[q, \nabla_{r,s}, k, j][q, \varepsilon, i, k] \vdash [q', \nabla_{r,s-1}, i, j] / q' \in \text{reveal}(q)\} \\
\mathcal{D}_{\text{init}} & = \{[q_0, \varepsilon, 0, 0]\} \\
\mathcal{D}_{\text{head}} & = \{[q, \nabla_{r,0}, i, j] \vdash [q', \varepsilon, i, j] / q' \in \text{goto}(q, A_r, 0)\}
\end{align*}
$$

with $q_0 \in Q$ the initial state, and $\text{action}$ and $\text{goto}$ entries in the PDA tables [1]. We say that $q' \in \text{reveal}(q)$ iff $\exists Y \in N \cup \Sigma$ such that $\text{shift}_{q} \in \text{action}(q', Y)$ or

$^1$ For Incremental Context-free Environment.

$^2$ The empty string, $\varepsilon$, is represented by the empty set of hypotheses, $\emptyset$. An input string $w_1\ldots w_n$, $n \geq 1$ is represented by $\{[w_1, 0, 1], [w_2, 1, 2], \ldots, [w_n, n-1, n]\}$.
q \in \text{goto}(q', Y)$, that is, when there exists a transition from $q'$ to $q$ in $A$. This set is equivalent to the dynamic interpretation of non-deterministic PDA’s:

- A deduction step $\text{Init}$ is in charge of starting the parsing process.
- A deduction step $\text{Shift}$ corresponds to pushing a terminal $a$ onto the top of the stack when the action to be performed is a shift to state $q'$.
- A step $\text{Sel}$ corresponds to pushing the $\nabla_{r,n_r}$ symbol onto the top of the stack in order to start the reduction of a rule $r$.
- The reduction of a rule of length $n_r > 0$ is performed by a set of $n_r$ steps $\text{Red}$, each of them corresponding to a pop transition replacing the two elements $\nabla_{r,s} X_{r,s}$ placed on the top of the stack by the element $\nabla_{r,s-1}$.
- The reduction of a rule $r$ is finished by a step $\text{Head}$ corresponding to a swap transition that recognizes the top element $\nabla_{r,0}$ as equivalent to the left-hand side $A_{r,0}$ of that rule, and performs the corresponding change of state.

These steps are applied until no further items can be generated. The splitting of reductions into a set of $\text{Red}$ steps allows us to share computations corresponding to partial reductions, attaining a worst case time (resp. space) complexity $O(n^3)$ (resp. $O(n^2)$) with respect to the length $n$ of the sentence $[9]$. The sentence is recognized iff the final item $[q_f, \nabla_{0,0}, 0, n + 1], q_f \in Q_f$, is generated.

3 The Error Repair Strategy

Our next step is to extend the standard parser with an error repair strategy. Given that we choose to work using the technique and terminology described in [10], we limit our description to the essential concepts; referring the reader to the original paper in order to get more details.

3.1 The Parsing Scheme

Following Mauney and Fischer in [5], we talk about the error in a portion of the input to mean the difference between what was intended and what actually appears in the input. In this context, a repair should be understood as a modification on the input string allowing the parse both, to recover the standard process and to avoid the phenomenon of cascaded errors, that is, errors precipitated by a previous erroneous repair diagnostic. This is, precisely, the goal of the notion of regional repair defined in [10], which we succinctly introduce now.

To begin with, we assume that we are dealing with the first error detected. We extend the initial structure of items, as a quadruple $[p, X, i, j]$, with an error counter $e$, resulting in a new structure of the form $[p, X, i, j, e]$. For each error item, defined from the fact that no action is possible from it when the lookahead is $w_i$, we investigate the list of its associated detection items; that is, those items representing the recognition of a terminal in the input string where we effectively locate the error. These detection items are located by using the back pointer, which indicates the input position where the last PDA action was applied. So,
we recursively go back into its ancestors until we find the first descendant of the last node that would have had to be reduced if the lookahead had been correct. Once the detection items have been fixed, we apply the following deduction steps from them:

\[ D_{\text{Shift count}} = \{ [q, \varepsilon, i, j, 0] \vdash [q', \varepsilon, j, j + 1, 0] \} \]

\[ D_{\text{Insert error}} = \{ [q, \varepsilon, i, j, 0] \vdash [q, \varepsilon, j, j, I(a)] \} \]

\[ D_{\text{Delete error}} = \{ [q, \varepsilon, i, j, 0] \vdash [q, \varepsilon, j, j + 1, D(w)] \} \]

\[ D_{\text{Replace error}} = \{ [q, \varepsilon, i, j, 0] \vdash [q, \varepsilon, j, j + 1, R(a)] \} \]

where \( I(a), D(a) \) and \( R(a) \) are the costs for insertion, deletion and replacement of \( a \in \Sigma \), respectively; and usually computed from a metric. This process continues until a repair applies a reduction covering both error and detection items. Once this has been performed on each detection item, we select the least-cost repairs and the parser goes back to standard mode. Error counters are added at the time of reductions, even when error mode is finished:

\[ D_{\text{Sel count}} = \{ [q, \varepsilon, i, j, e] \vdash [q, \nabla_r, n, j, j, e] \} \]

\[ D_{\text{Red count}} = \{ [q, \nabla_r, s, k, j, e] \vdash [q', \varepsilon, i, k, e'] \} \]

\[ D_{\text{Head count}} = \{ [q, \nabla_r, 0, i, j, e] \vdash [q', \varepsilon, i, j, e] \} \]

To avoid the generation of items only differentiated by the error counter we shall apply a principle of optimization, saving only for computation purposes those items with minimal error counters \( e \).

When the current repair is not the first one, it can modify a previous repair in order to avoid cascaded repairs by adding the cost of the new error hypotheses to profit from the experience gained from previous ones. This allows us to get, in a simple manner, an asymptotic behavior close to global repair methods [10]. As a consequence, although time (resp. space) complexity is, in the worst case, \( O(n^3) \) (resp. \( O(n^2) \)), in practice it is considerably more reduced as we shall see in our experimental tests. The input string is recognized iff the final item \([q_f, \nabla_0, 0, n + 1, e], q_f \in Q_f \), is generated.

### 3.2 Previous Works

Error recovery methods can be classified into local, global and regional strategies. Local repair algorithms [2, 6] make modifications to the input so that at least one more original input symbol can be accepted by the parser. There are cases, however, in which their simplicity causes them to choose a poor repair.

Global algorithms [4] examine the entire program and make a minimum of changes to repair all the errors. Global methods give the best repairs possible, but they are not efficient. Since they expend equal effort on all parts of the program, including areas that contain no errors, much of that effort is wasted. Finally, the main problem to be dealt with in regional approaches [5] is how to determine the extent of the repair in order to avoid cascaded errors.

Robust Parsing Using Dynamic Programming 261
Our proposal is a least-cost regional error repair strategy, asymptotically equivalent to global repair ones. That is, in the worst case, space and time complexity are the same as those attained for global repairs and, in the best case, are the same as for local ones. The repair quality is equivalent to global approaches. In relation to local ones, no fixed repair region is considered. So, we avoid both wasted effort when it is excessive and the generation of cascaded errors when it is not sufficient. Compared to other regional algorithms [5], we provide a least-cost dynamic estimation of this region, which is an advantage in the design of interactive tools, where efficiency is a priority challenge.

4 Parsing Incomplete Sentences

In order to handle incomplete sentences, we extend the input alphabet. We introduce two symbols, "?" stands for one unknown word symbol, and "∗" stands for an unknown sequence of input word symbols.

4.1 The Parsing Scheme

Once the parser detects that the next input symbol to be shifted is one of these two extra symbols, we apply the set of deduction steps $D_{\text{incomplete}}$, which includes the following two sets of deduction steps, as well as $D_{\text{count}}$ previously defined:

$$D_{\text{Shift incomplete}} = \{ [q, \varepsilon, i, j] \vdash [q', \varepsilon, j, j + 1] \mid \exists [?, j, j + 1] \in \mathcal{H} \}$$

$$D_{\text{Loop shift incomplete}} = \{ [q, \varepsilon, i, j] \vdash [q', \varepsilon, j, j] \mid \exists [\ast, j, j + 1] \in \mathcal{H} \}$$

From an intuitive point of view, $D_{\text{Shift incomplete}}$ applies any shift transition independently of the current lookahead available, provided that this transition is applicable with respect to the PDA configuration and that the next input symbol is an unknown token. In relation to $D_{\text{Loop shift incomplete}}$, it applies to items corresponding to PDA configurations for which the next input symbol denotes an unknown sequence of tokens, any valid shift action on terminals or variables. Given that in this latter case new items are created in the same starting itemset, shift transitions may be applied any number of times to the same computation thread, without scanning the input string.

All deduction steps are applied until every parse branch links up to the right-context by using a shift action, resuming the standard parse mode. In this process, when we deal with sequences of unknown tokens, we can generate nodes deriving only "∗" symbols. This over-generation is of no interest in most practical applications and introduces additional computational work, which supposes an extra loss of parse efficiency. So, our goal is to replace these variables with the unknown subsequence terminal, "∗". We solve this problem by re-taking the counters introduced in error repair mode, in order to tabulate the
number of categories used to rebuild the noisy sentence. In effect, our final goal is to select, for a given ill-formed input, an optimal reconstruction. As a consequence, it makes no sense to differentiate between counter contributions due to the application of one or another parsing mechanism. When several items representing the same node are generated, only those with minimal counter are saved. Formally, we redefine the set of deduction steps as follows:

\[
D_{\text{incomplete}}^{\text{Shift}} = \{(q, \varepsilon, i, j, e) \vdash (q', \varepsilon, j, j + 1, e + I(a)) \mid \exists [?, j, j + 1] \in H \quad \text{shift}_{q'} \in \text{action}(q, a) \}
\]

\[
D_{\text{incomplete}}^{\text{Loop-shift}} = \{(q, \varepsilon, i, j, e) \vdash (q', \varepsilon, j, j, e + I(X)) \mid \exists [* , j, j + 1] \in H \quad \text{shift}_{q'} \in \text{action}(q, X) \quad X \in N \cup \Sigma
\]

where \( I(X) \) is the insertion cost for \( X \in N \cup \Sigma \), and we maintain the definition domain previously considered for \( D_{\text{Red}}^{\text{count}}, D_{\text{Sel}}^{\text{count}} \) and \( D_{\text{Head}}^{\text{count}} \). The incomplete sentence is recognized iff \( [q_f, \nabla_{0,0}, 0, n + 1, e], q_f \in Q_f \) is generated.

4.2 Previous Works

Previous proposals, such as Tomita et al. [8] and Lang [3], also apply dynamic programming to deal with unknown sequences in order to reduce space and time complexity, although the approach is different in each case. From an operational point of view, Lang introduces items as fragments of the PDA computations that are independent of the initial content of the stack, except for its two top elements. This relies on the concept of dynamic frame for \( CFG \)'s [9] and, in particular, to the dynamic frame \( S^2 \). Tomita et al. use a shared-graph based structure to represent the stack forest. We work in a dynamic frame \( S^1 \), which means that items only represent the top of the stack. This results in improved sharing for both syntactic structures and computations.

In relation to the parsing scheme applied, Lang separates the execution strategy from the implementation of the interpreter, while Tomita et al.'s work can be interpreted simply as a specification of Lang's for LR(0) PDA’s. We consider a LALR(1) scheme, which facilitates lookahead computation, whilst the state splitting phenomenon remains reasonable. This enables us to achieve high levels of sharing and efficiency as well as to increase the deterministic domain.

Neither Lang nor Tomita et al. avoid over-generation in nodes deriving only "*" symbols. Only Lang includes an additional phase to eliminate these nodes from the output parse shared forest. We solve both the consideration of an extra simplification phase and the over-generation on unknown sequences by considering the same principle of optimization applied on error counters in the error repair process.

5 The Robust Parser

We are now ready to introduce the robust parsing construction. In order to favor understanding, we differentiate two kinds of parse steps. So, we talk about
extensional steps when we deal with deduction steps including conditions over shift actions in standard parsing mode, and we talk about intensional steps in any other case, i.e. when they are related to reduce actions in the kernel. Whichever is the case, the robust mode must guarantee the capacity to recover the parser from any unexpected situation derived from either gaps in the scanner or errors. To deal with this, it is sufficient to combine the deduction steps previously introduced. More exactly, we have that the extensional steps are defined by:

\[ D^{Init} \cup D^{Shift}\_count \cup D^{Insert} \cup D^{Delete} \cup D^{Replace} \cup D^{Shift\_error} \cup D^{Delete\_error} \cup D^{Replace\_error} \cup D^{Shift\_incomplete} \cup D^{Delete\_incomplete} \]

and the intensional ones by

\[ D^{Red\_count} \cup D^{Sel\_count} \cup D^{Head\_count} \]

where there is no overlapping between the deduction subsets. In effect, in relation to the extensional case, no collision is possible because the steps in question are distinguished by conditions over the lookahead. For the intensional case, the steps remain invariable from the beginning, when we defined the standard parser. The final robust parse has a time (resp. space) complexity, in the worst case, \( O(n^3) \) (resp. \( O(n^2) \)) with respect to the length \( n \) of the ill-formed sentence. The input string is recognized iff the final item \([q_f, \nabla_{0,0}, 0, n+1, e], q_f \in Q_f\), is generated.

6 Experimental Results

We consider the language, \( \mathcal{L} \), of arithmetic expressions to illustrate our discussion, comparing the standard parsing on ICE \([9]\), with the consideration of full robust parsing. We introduce two grammars, \( \mathcal{G}_L \) and \( \mathcal{G}_R \), given by the rules:

\[
\mathcal{G}_L: E \rightarrow E + T \mid T \\
T \rightarrow (E) \mid \text{number}
\]

\[
\mathcal{G}_R: E \rightarrow T + E \mid T \\
T \rightarrow (E) \mid \text{number}
\]

to generate the running language, \( \mathcal{L} \). As a consequence, parses are built from the left-associative (resp. right-associative) interpretation for \( \mathcal{G}_L \) (resp. \( \mathcal{G}_R \)), which allows us to estimate the impact of traversal orientation in the parse process. Our goal now is essentially descriptive, in order to illustrate the recovery mechanisms and its behavior in a variety of situations. In this context, our example joins structural simplicity and topological complexity in a language which is universally known. In the same sense, larger languages do not provide extra criteria to be considered.

In this sense, we shall consider four different patterns to model ill-formed input strings. The former, that we shall call error-correction, is of the form

\[ b_1 + \ldots + b_{i-1} + (b_i + \ldots + (b_{[n/3]} + b_{[n/3]+1}b_{[n/3]+2} + \ldots + b_{t}b_{t+1} + b_{t+2} + \ldots + b_n \]

The second, that we shall call unknown, is of the form

\[ b_1 + \ldots + b_{i-1} + (b_i + \ldots + (b_{[n/3]} + b_{[n/3]+1} \ast + b_{[n/3]+3} \ast + \ldots + b_{t} \ast + b_{t+2} + \ldots + b_n \]
The third pattern, that we shall call *total overlapping*, is of the form

\[ b_1 + \ldots + b_{i-1} + (b_i + \ldots + (b_{\lfloor n/3 \rfloor} + \ast b_{\lfloor n/3 \rfloor + 1}b_{\lfloor n/3 \rfloor + 2} + \ldots + \ast b_{\ell + 1}b_{\ell + 2} + \ldots + b_n) \]

The last pattern, that we shall call *partial overlapping*, is of the form

\[ b_1 + \ldots + b_{i-1} + (b_i + \ldots + (b_{\lfloor n/3 \rfloor} + b_{\lfloor n/3 \rfloor + 1}b_{\lfloor n/3 \rfloor + 2} + \ldots + b_{\ell + 1} \ast b_{\ell + 2} \ldots \ast b_n) \]

where \( i \in \{\lfloor n/3 \rfloor, \ldots, 1\} \) and \( \ell = 3\lfloor n/3 \rfloor - 2i + 1 \), with \( \lfloor n/3 \rfloor \) being the integer part of \( n/3 \). As distance to compute the error costs we consider, in this case, the discrete metric. This implies that these costs are zero on scan actions, and one when a insertion, deletion or replacement hypothesis is applied.

Our running examples seek to illustrate the variety of situations to be dealt with in robust parsing. So, the *unknown* example only requires the treatment of unknown sequences, while the *error-correction* example only applies the error repair strategy. The *total overlapping* example forces the system to apply both unknown sequences recognition and error repair, although only the error recovery mechanisms are finally taken into account. Finally, the *partial overlapping* example also combines the recognition of unknown sequences and error repair, but in this case both strategies have an active role in the parse recovery process.
In the case of unknown pattern, the set of minimal cost robust parse process includes the sentences obtained by inserting closed brackets in the positions indicated by the unknown sequence. In other patterns, the set of minimal cost robust parse process is formed by the sentences obtained by replacing tokens \( b_{\lfloor n/3 \rfloor + 2k} \) with \( k \in \{1, \ldots, \lfloor n/3 \rfloor - i + 1 \} \) by closed brackets. As a consequence, one minimal cost parse alternative is given by "\( b_1 + \ldots + b_i + b_{i-1} + (b_i + \ldots + (b_{\lfloor n/3 \rfloor} + b_{\lfloor n/3 \rfloor + 1}) + \ldots + b_\ell) + b_{\ell+2} + \ldots + b_n \)" whose parse cost we shall use as reference to illustrate the practical complexity of our proposal in these experimental tests.

The results are shown, for the unknown and error-correction examples, in Fig. 1, in the left and the right-hand-side of the figure respectively. In the case of total and partial overlapping examples, the tests are shown in the left and the right-hand-side of Fig. 2. In all cases, the results are provided for both grammars \( G_L \) and \( G_R \), with the number of items generated by the system during the parse process being taken as a reference for appreciating the efficiency of our method, rather than purely temporal criteria, which are more dependent on its implementation. These items are measured in relation to the position, \( \ell \), of the addend "\( b_\ell \)" in the input string, around which all the tests have been structured.

The first detail to note is the null slopes in the graphs of the total overlapping example, while for all the others the slopes are ascendent. This is due to the particular distribution of the zone where we apply the robust parse mechanisms. In effect, as is shown in Fig. 3, the error detection points from the very first point of error in the input string locate the beginning of the error correction zone \([10]\) at the addend "\( b_1 \)". In practice, as part of the more general robust parse process, the error correction strategy already covers all the input string, although only in the case of \( G_R \) does the error repair scope extend to global context. This apparent contradiction in the case of \( G_L \) is due to the fact that although the effective repair mechanisms do not have a global scope, most unsuccessful repairs are only rejected at the end of the robust parse process. As a consequence, for both grammars in this example the correction mechanisms are applied on all the input positions, and the location of "\( b_\ell \)" has no influence on the number of items generated, as can be seen in Fig. 2. This is also illustrated in Fig. 4, representing on its left-hand-side the increase in the size of the repair scope for both error


The situation is different in the error correction and partial overlapping examples, for which the size of the error repair zone increases with the position of “by”, as is shown in Fig. 5. In this sense, the figure illustrates both the dependence of the error repair region on the grammar used, and the asymptotic behavior of the error repair strategy \[10\] in dealing with cascaded errors.

In relation to complexity, although the theoretical cost is the same for both the error repair strategy and the treatment of unknown sentences, in practice these tests show that the greater weight is due to error repair. This is illustrated by the results displayed for the error correction and the two overlapping examples on the right-hand-sides of Fig. 1 and Fig. 2, respectively. In effect, these results show that the amount of items generated is appreciably larger in these cases, in contrast to the work developed for the unknown example, which we can see in the left-hand-side of Fig. 1, and for which no error repair process is applied. From an operational point of view, this behavior is a consequence of the contextual treatment in each case. So, the parse of unknown sequences only generates, for each symbol \(\ast\), items in the current itemset. However, in the case of error repair the scope depends, for each error, on the grammatical structure and can range from one to the total collection of itemsets, as is shown in Figs. 3 and 5.

\[
\text{points of detection:} \quad \begin{array}{c}
\text{first point of error, either case} \quad \text{second point of error, } G_L \text{ case} \quad \text{second point of error, } G_R \text{ case} \quad \ldots \\
\end{array}
\]

\[
\ldots + \left( b_{p_{m/3}-1} + b_{p_{m/3}} + b_{p_{m/3+1}} + b_{p_{m/3+2}} + b_{p_{m/3+3}} + b_{p_{m/3+4}} + \ldots \right)
\]

\[\text{Fig. 5. Error detection points for the error correction and partial overlapping examples}\]
Whichever is the case, the smoothness of the slopes proves the computational efficiency of our proposal.

7 Conclusions

Robust parsing is a central task in the design of dialogue systems, where the deterioration of the signal, and the presence of under-generation or over-generation phenomena due to covering grammatical problems make apparently impossible to perform continuous unrestricted language recognition.

In this sense, robust parsing seeks to find those interpretations that have maximal thresholds. Our proposal provides the capacity to recover the system from external syntactic factors or user errors; and also the possibility to do so efficiently. Here, our work concentrates on enhancing robustness by using the mechanisms offered by dynamic programming in order to improve performance and provide an optimal quality in the parse definition.

References


LR Parsing for Global Index Languages (GILs)

José M. Castaño
Computer Science Department
Brandeis University
jcastano@cs.brandeis.edu

Abstract. We present here Global Index Grammars (GIGs) and the characterizing 2 Stack automaton model (LR-2PDA). We present the techniques to construct an LR parsing table for deterministic Global Index Grammars. GILs include languages which are beyond the power of Linear Indexed Grammars/Tree Adjoining Grammars. GILs generalize properties of CF Languages in a straightforward way and their descriptive power is relevant at least for natural language and molecular biology phenomena.

1 Introduction

Context-free grammars (CFGs) is perhaps the most developed, well understood and most applicable part of formal language theory. Many formalisms have been proposed to extend the power of context-free grammars using control devices, where the control device is a context-free grammar (see [23] and [11] for control languages). In an automaton model those extensions either use constrained embedded or additional stacks (cf. [4], [21]). The appeal of this approach is that many of the attractive properties of context-free languages are inherited (e.g. polynomial parsability, semilinearity, closure properties).

Those models can be generalized such that additional control levels, additional levels of embeddedness, or additional stacks can be added. They form hierarchies of levels of languages, where a language of level $k$ properly includes a language of level $k−1$ (we refer to $k$–level languages in [23] sense, but see also,[16], [18], [9]). Those generalizations provide more expressive power but at a computational cost. The complexity of the recognition problem is dependent on the language level: for a level-$k$ language it is in $O(n^{3·2^{k−1}})$.

In [6] we introduced Global Index Grammars (GIGs) - and GILs the corresponding languages - as an alternative grammar formalism that has a restricted context sensitive power. We showed that GIGs have enough descriptive power to capture phenomena relevant for natural languages ([11]) and molecular biology ([17]) which are beyond context free: reduplication or repeats, multiple agreements, crossed agreements or pseudo-knots. It includes such languages as $\{ww^+ | w \in \Sigma^*\}$ or $\{a^n b^m (c^n d^m)^+ | n, m \geq 1\}$ which are also beyond the power of Tree Adjoining Languages and beyond the power of any level-$k$ control language. Recognition of the language generated by a GIG is a bounded polynomial time. In [6] we presented an Earley type recognition algorithm for
GIGs that performs in $O(n^6)$. However the time complexity for bounded state grammars with unambiguous indexing is $O(n)$. In [8] we discuss the relevance of GIGs for natural language phenomena in terms of the structural descriptions GIGs can generate. The goal of this paper is to present the equivalent automaton model which extends in a natural way properties of PDAs and context-free languages. Due to space constraints the characterization theorems of GILs using the automaton model are not included in this paper. The characterization theorems, a more detailed comparison of this automaton with PDAs and Turing Machines is available at [7]. The automaton model and the grammar can be used to prove that the family of GILs is an abstract family of languages (AFL) (cf. [7]). GILs have also the semilinear property: a proof can be easily built following [13] (see [7]). In this paper we present a technique to construct a parsing table for an LR parsing algorithm for GILs. Deterministic LR parsing techniques are relevant, for example, for higher level programming languages or deterministic approaches for shallow parsing in natural language. They are also the basis for non-deterministic approaches, such as a non-deterministic Generalized LR parsing. The construction of the Goto function shows an interesting (though not surprising) fact about GILs: the Goto function defines a PDA that recognizes the set of viable prefixes.

2 Global Index Grammars

2.1 Linear Indexed Grammars

Indexed grammars, (IGs) [1], and Linear Index Grammars, (LIGs; LILs) [12], have the capability to associate stacks of indices with symbols in the grammar rules. IGs are not semilinear. LIGs are Indexed Grammars with an additional constraint in the form of the productions: the stack of indices can be “transmitted” only to one non-terminal.

A Linear Indexed Grammar is a 5-tuple $(V, T, I, P, S)$, where $V$ is the set of variables, $T$ the set of terminals, $I$ the set of indices, $S$ in $V$ is the start symbol, and $P$ is a finite set of productions of the form, where $A, B \in V$, $\alpha, \gamma \in (V \cup T)^*$, $i \in I$:

a. $A[..] \rightarrow \alpha B[..] \gamma$  
   b. $A[i..] \rightarrow \alpha B[..] \gamma$  
   c. $A[..] \rightarrow \alpha B[i..] \gamma$

2.2 Global Indexed Grammars

GIGs use the stack of indices as a unique global control structure. This formalism provides a global but restricted context that can be updated at any local point in the derivation. GIGs are a kind of regulated rewriting mechanisms [10] with global context and history of the derivation (ordered derivation) as the main characteristics of its regulating device.

Definition 1. A GIG is a 6-tuple $G = (N, T, I, S, #, P)$ where $N, T, I$ are finite pairwise disjoint sets and 1) $N$ are non-terminals 2) $T$ are terminals 3) $I$ a set of stack indices 4) $S \in N$ is the start symbol 5) $#$ is the start stack symbol (not
LR Parsing for Global Index Languages (GILs) 271

in $I,N,T$ and 6) $P$ is a finite set of productions, having the following form,\(^1\)
where $x \in I$, $y \in \{I \cup \#\}$, $A \in N$, $\alpha, \beta \in (N \cup T)^*$ and $a \in T$.

a. $A \rightarrow \epsilon$ (epsilon or context-free rules) or the equivalent $A \rightarrow \alpha$

b. $A \rightarrow a \beta$ (push) $[\ldots]A \rightarrow [x..]a \beta$

c. $A \rightarrow \bar{x} \alpha$ (pop) $[\ldots]A \rightarrow [\ldots]\alpha$

Note that push (type b) productions require the right-hand side of the rule to contain a terminal in the first position. This constraint on push rules is a crucial property of GIGs. It has a correlate in the automaton model as we will see in section 3. Derivations in a GIG are similar to those in a CFG except that it is possible to modify a string of indices. We define the derives relation $\Rightarrow$ on sentential forms, which are strings in $I^*#(N \cup T)^*$ as follows. Let $\beta$ and $\gamma$ be in $(N \cup T)^*$, $\delta$ be in $I^*$, $x$ in $I$, $w$ be in $T^*$ and $X_i$ in $(N \cup T)$.

1. If $A \rightarrow X_1...X_k$ is a production of type (a.) (i.e. $\mu = \epsilon$ or $\mu = [x]$, $x \in I$) then:
   $$\delta#\beta A \gamma \Rightarrow \delta#\beta X_1...X_k \gamma$$ (if $\mu = [x]$ then $\delta = x\delta_1$)

2. If $A \rightarrow a X_1...X_n$ is a production of type (b.) or push: $\mu = x, x \in I$, then:
   $$\delta#w A \gamma \Rightarrow x\delta#waX_1...X_n \gamma$$

3. If $A \rightarrow X_1...X_n$ is a production of type (c.) or pop : $\mu = \bar{x}, x \in I$, then:
   $$x\delta#w A \gamma \Rightarrow \delta#wX_1...X_n \gamma$$

The reflexive and transitive closure of $\Rightarrow$ is denoted, as usual by $\Rightarrow^*$. We define the language $L$ of a GIG, $G$, $L(G)$ to be: \{w|$\#S \Rightarrow^* \#w$ and $w$ is in $T^*$\} and we call $L$ a GIL.

The main difference between, IGs, LIGs and GIGs, corresponds to the interpretation of the derives relation relative to the behavior of the stack of indices. In IGs the stacks of indices are distributed over the non-terminals of the right-hand side of the rule. In LIGs, indices are associated with only one non-terminal at right-hand side of the rule. This produces the effect that there is only one stack affected at each derivation step, with the consequence of the semilinearity property of LILs. GIGs share this uniqueness of the stack with LIGs: there is only one stack to be considered. Unlike LIGs and IGs the stack of indices is independent of non-terminals in the GIG case. GIGs can have rules where the right-hand side of the rule is composed only of terminals and affect the stack of indices. Indeed push rules (type b) are constrained to be in Greibach Normal Form as specified in (6.b) in the GIG definition. The derives definition requires

\(^1\) The notation in the rules makes explicit that operation on the stack is associated to the production and neither to terminals nor to non-terminals. It also makes explicit that the operations are associated to the computation of a Dyck language (using such notation as used in e.g. [14]).
a leftmost derivation for those rules that affect the stack of indices (push and pop rules).

The following examples show that GILs contain languages which are not CFLs nor Linear Indexed Languages nor any level−k control language (in sense) for any k or any m−Multiple Context Free Language for any m [18], as can be easily proven using the corresponding pumping lemmas. The first example is crucial to model coordination phenomena in natural language (cf. [12]) and multiple repeats in biology.

Example 1 (Multiple Copies). \(L(G_{wwn}) = \{ww^n \mid w \in \{a, b\}^*\}\)
\[G_{wwn} = (\{S, R, A, B, C, L\}, \{a, b\}, \{i, j\}, S, \#, P)\] and where \(P\) is:
\[
\begin{align*}
S & \rightarrow AS \mid BS \\
C & \rightarrow RC \\
R & \rightarrow RA \\
R & \rightarrow RB \\
R & \rightarrow \epsilon
\end{align*}
\]
\[
\begin{align*}
A & \rightarrow a \\
B & \rightarrow b \\
L & \rightarrow La \mid a \\
L & \rightarrow Lb \mid b
\end{align*}
\]
The derivation of \(ababab\):
\[
\begin{align*}
#S & \Rightarrow #AS \\
i & \Rightarrow i##aS \\
ji & \Rightarrow ji#abS \\
ji & \Rightarrow ji#abRC \\
ji & \Rightarrow #abRABC \\
ji & \Rightarrow i#abaBC \\
ji & \Rightarrow #ababLb \\
# & \Rightarrow abab
\end{align*}
\]

Example 2 (Multiple dependencies). \(L(G_{gdp}) = \{a^n(b^n c^n)^+ \mid n \geq 1\}\)
\[G_{gdp} = (\{S, A, R, E, O, L\}, \{a, b, c\}, \{i\}, S, \#, P)\] and \(P\) is:
\[
\begin{align*}
S & \rightarrow AR \\
A & \rightarrow aAE \\
E & \rightarrow a \\
E & \rightarrow b \\
R & \rightarrow b L
\end{align*}
\]
\[
\begin{align*}
L & \rightarrow OR \mid C \\
C & \rightarrow cC \mid c \\
O & \rightarrow cOE \mid c
\end{align*}
\]
The derivation of the string \(aabbccbbcc\) shows five dependencies.
\[
\begin{align*}
#S & \Rightarrow #AR \\
# & \Rightarrow #aAER \\
# & \Rightarrow #aaER \\
# & \Rightarrow #aabL \\
# & \Rightarrow #aab OR \\
# & \Rightarrow #aabccOER \\
# & \Rightarrow #aabccER \\
# & \Rightarrow #aabcbRii \\
# & \Rightarrow #aabccbbL \\
# & \Rightarrow #aabccbbC \\
# & \Rightarrow #aabccbbcc
\end{align*}
\]

3 The Two Stack Automaton Model: LR-2PDA

Different models of 2-Stack Push Down Automata have been proposed, (e.g. [22], [5], also generalized models of multistacks (e.g. [4], [9]). The model of 2-PDA we present here (a more detailed presentation can be found in [7]) preserves the main characteristics of a PDA but has a restricted additional storage capacity: it enables access to some limited left context information. It behaves like a PDA, in the sense that it is forced to consume the input from left to right and has a limited capacity to go back from right to left. Consequently it was called LR-2PDA. We need to guarantee that the proposed automaton model does not have the power of a Turing Machine. The restrictions to the transitions performed by the kind of 2-PDA presented in [7] are as follows: the Main stack behaves like an ordinary PDA stack (same type of transitions that a PDA allows are allowed). Transitions that push a symbol into the second auxiliary stack do not allow \(\epsilon\) moves. It can be observed the similarity between the restrictions on GIG productions and LR-2PDA transitions, in particular between type b. transitions and productions. We show in [7] that LR-2PDAs characterize GI languages.
Definition 2. A LR-2PDA is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, \bot, F)\) where \(Q, \Sigma, \Gamma, \) and \(F\) are all finite sets and 1. \(Q\) is the set of states, 2. \(\Sigma\) is the input alphabet, 3. \(\Gamma\) is the stacks alphabet, 4. \(\delta\) is the transition function (where \(\mathcal{P}_F\) denotes a finite power set):
- a. \(Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \{\epsilon\} \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \{\epsilon\})\) and
- b. \(Q \times \Sigma \times \Gamma \times \Gamma \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \Gamma^+)\)
- c. \(Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times \Gamma \rightarrow \mathcal{P}_F(Q \times \Gamma^* \times \{\epsilon\})\)

5. \(q_0 \in Q\) is the start state 6. \(\bot \in \Gamma \setminus \Sigma\) is the bottom marker of the stacks and 7. \(F \subseteq Q\) is the set of accept states.

The instantaneous descriptions (IDs) of a LR-2PDA describe its configuration at a given moment. They specify the current input, state and stack contents. An ID is a 4-tuple \((q, w, \gamma_1, \gamma_2)\) where \(q\) is a state \(w\) is a string of input symbols and \(\gamma_1, \gamma_2\) are strings of stack symbols. If \(M = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)\) is a LR-2PDA, then \((q, aw, Z\alpha, O\beta) \vdash_M (p, w, \zeta\alpha, \eta\beta)\) if \(\delta(q, a, Z, O)\) contains \((p, \zeta, \eta)\). The reflexive and transitive closure of \(\vdash_M\) is denoted by \(\vdash_M^*\). The language accepted by a LR-2PDA \(M = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)\) is \(\{w \mid (q_0, w, \bot, \bot) \vdash_M^* (p, \epsilon, \bot, \bot)\}\) for some \(p\) in \(F\) (acceptance by empty stacks).

A deterministic LR-2PDA is defined as follows:

Definition 3. An LR-2PDA is deterministic iff:

1. for each \(q \in Q\) and \(Z, I \in \Gamma\), whenever \(\delta(q, \epsilon, Z, I)\) is nonempty, then \(\delta(q, a, Z, I)\) is empty for all \(a \in \Sigma\).
2. for no \(q \in Q\), \(Z, I \in \Gamma\), and \(a \in \Sigma \cup \{\epsilon\}\) does \(\delta(q, a, Z, I)\) contain more than one element.
3. for each \(q \in Q\), \(A \in \Sigma\) and \(Z, I \in \Gamma\), whenever \(\delta(q, A, Z, \epsilon)\) is nonempty, then \(\delta(q, A, Z, I)\) is empty for all \(I \in \Gamma\).

Condition 1 prevents the possibility of a choice between a move independent of the input symbol (\(\epsilon\)-move) and one involving an input symbol.
Condition 2 prevents the choice of move for any equal 4-tuple.
Condition 3 prevents the possibility of a choice between a move independent of the second stack symbol (\(\epsilon\)-move) and one involving a symbol from the stack.

We can regard \(\epsilon\) to be an abbreviation for all the possible symbols in the top of the stack. Therefore, the first case is not compatible with the third, nor with the fourth.

1. \((q, a, \epsilon, \epsilon) = (q, \epsilon, i)\) 2. \((q, b, \epsilon, \epsilon) = (q, \epsilon, j)\)
3. \((q, a, \epsilon, i) = (q, \epsilon, \epsilon)\) 4. \((q, a, \epsilon, j) = (q, \epsilon, j)\)

The similarities of Earley Parsing and LR parsing are made explicit in transformations from Earley Parsing Schema to LR Parsing Schema (cf. [19]). Due to space constraints we are not able to compare the Earley algorithm we presented in [6] with the LR approach we present here. The extension we present here is not a straightforward conversion of the Earley algorithm for GIGs.
4 LR Parsing for Deterministic GILs

An LR parser for a CFL is essentially a compiler that converts an LR CFG into a DPDA automata (cf. [15], [2]). In the GIG case, the technique we present here converts an LR GIG into a deterministic LR-2PDA, according to the definition given above. This approach can be extended to a Generalized LR parsing approach, if multiple values are allowed in the parsing table (cf. [20]).

4.1 LR Items and Viable Prefixes

We assume the reader has knowledge of LR parsing techniques (cf. [2]). The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes in context free LR-parsing. An item $A \rightarrow \beta_1 \cdot \beta_2$ is valid for a viable prefix $\alpha \beta_1$ if there is a derivation $S' \Rightarrow^* \alpha A \Rightarrow^* \alpha \beta_1 \beta_2 w$.

We keep this notion of viable prefixes to describe the computation of the main stack in a LR-2PDA. The notion of valid items for GILs is more complex because a GIG LR-item has to encode the possible configurations of the auxiliary stack. It is also more complex because GIL derivation requires leftmost derivation for the stack operations.

We say a GIG item index pair $[\delta, A \rightarrow \mu \beta_1 \cdot \beta_2]$ where $\delta$ is in $I \cup \{\#\}$ is valid for a GIG viable prefix $\alpha \beta_1$ if there is a derivation using the GIG context free backbone: $S' \Rightarrow^* \alpha A \Rightarrow^* \alpha \beta_1 \beta_2 w$ and there is a GIG derivation $S' \Rightarrow \gamma i_1 a_1 ... a_i \rightarrow A \beta \Rightarrow^* \delta \gamma j a_1 ... a_j \beta_2 \beta$.

Therefore the set of items has to encode the corresponding top-down prediction that affects the stack of indices. We will do this in a way that the set of items contain information not only of the possible items that belong to the set but also of the possible indices that might be associated to the set of parsing configurations that the items describe. The sets that define the states in GIG LR parsing are composed by a pair of sets $(IT, IN)$.

4.2 The Closure Operation

The Closure operation is equivalent to the Predict function in Earley parsing (cf.[3]). However in Earley’s algorithm this computation is done in run-time while in LR parsing it is precompiled in the LR states. In the GIG case the set of items and possible indices values cannot be determined only by the Closure operation. They are also determined by the Goto operation. If $IT$ is a set of items for a grammar $G$, then $\text{closure}(IT)$ is the set of items constructed from $IT$ by the two rules:

1. Initially, every item in $IT$ is added to $\text{closure}(IT)$.
2. If $A \rightarrow \alpha \cdot B \beta$ is in $\text{closure}(IT)$ and i) $B \rightarrow \gamma$ is a production where $\delta$ is in $I \cup \{\epsilon\}$, or ii) $B \rightarrow a \cdot \gamma$ then add the item $B \rightarrow \delta \cdot \gamma$ (case 1) or the item $B \rightarrow \delta \cdot a \cdot \gamma$ (case 2) to $IT$ if it is not already there. Apply this rule until no more new items can be added to the closure of $IT$. 
Note that the two conditions in the Closure operation rule out the possible index operations $\overline{\delta}$ and $[\delta]$ when the right-hand side of the production starts with a non-terminal. Those productions are considered in case 2 of the Goto function.

In the CFG case (cf. [15]) the Closure operation is equivalent to obtain the set of states of a NFA that recognizes the viable prefixes. The transitions of this NFA are of the following form:

$$\delta(A \rightarrow \alpha \cdot B \beta, \epsilon) = \{ B \rightarrow \ast \gamma \mid B \rightarrow \gamma \text{ is a production } \}$$ (a NFA transition)

In the GIG case the transitions should be as follows, if $\overline{i}$ is the index operation of the production (PDA transitions):

$$\delta(A \rightarrow \alpha \cdot B \beta, \epsilon, \overline{i}) = \{ (B \rightarrow \ast \overline{i} \cdot \gamma, \epsilon) \mid B \rightarrow \ast \overline{i} \gamma \text{ is a production } \}$$ (PDA transit.)

It is apparent that both items should not be conflated together because there is a change in the stack configuration.

We will use as an example the grammar for the language $L_{wcw}$ to depict the construction of the set of items, to goto function and the parsing table.

Example 3.

$L(G_{wcw}) = \{wcw \mid w \in \{a, b\}^*\}$, $G_{wcw} = (\{S, R\}, \{a, b\}, \{i, j\}, S, \#, P)$ and $P$ is:

1. $S \rightarrow i \cdot a \cdot S$  
2. $S \rightarrow j \cdot b \cdot S$  
3. $S \rightarrow c \cdot R$
4. $R \rightarrow \overline{i} \cdot Ra$  
5. $R \rightarrow \overline{j} \cdot Rb$  
6. $R \rightarrow \epsilon$

The set of states for the grammar $G_{wcw}$ should encode the possible configurations of the stack of indices as follows:

$I_0$: $\# \rightarrow \cdot S$
$I_1$: $i \rightarrow a \cdot S$
$I_2$: $j \rightarrow b \cdot S$
$I_3$: $\{i, j, \#\} \rightarrow S \rightarrow c \cdot R$
$I_4$: $\{i, j, \#\} \rightarrow S \rightarrow i \cdot a \cdot S$
$I_5$: $\{i, j, \#\} \rightarrow S \rightarrow j \cdot b \cdot S$
$I_6$: $\{i, j, \#\} \rightarrow R \rightarrow \epsilon$
$I_7$: $\# \rightarrow j \cdot b \cdot S$
$I_8$: $\# \rightarrow i \cdot a \cdot S$
$I_9$: $\# \rightarrow c \cdot R$
$I_{10}$: $\# \rightarrow R \rightarrow \epsilon$
$I_{11}$: $\# \rightarrow j \cdot b \cdot S$
$I_{12}$: $\# \rightarrow j \cdot b \cdot S$
$I_{13}$: $\# \rightarrow j \cdot b \cdot S$

**Fig. 1.** Set of States for $G_{wcw}$

In the same way that the dot implies that some input symbols have been consumed, the new sets (as compared to context free sets) $I_4$ and $I_5$ imply that an index $i$ or $j$ has been consumed from the stack. The set $I_6$ requires that the stack be empty.
4.3 The Goto Operation

The CF function \( \text{goto}(IT, X) \), where \( IT \) is a set of items and \( X \) is a grammar symbol, is defined to be the closure of the set of all items \( [A \rightarrow \alpha X \cdot \beta] \) such that \( [A \rightarrow \alpha \cdot X \beta] \) is in \( IT \). Intuitively, if \( I \) is the set of items that are valid for some viable prefix \( \gamma \), then \( \text{goto}(I, X) \) is the set of items that are valid for the viable prefix \( \gamma X \). The context free Goto function is a DFA transition function where \( IT \) is in the state set and \( X \) is in the vocabulary.

In the GIG case, if \( I \) is the set of items that are valid for some viable prefix-index pairs \( (\gamma, \iota) \), then \( \text{goto}(I, X, \iota) = (I_j, \iota_j) \) is the set of items that are valid of the viable prefix-index pairs \( (\gamma X, \iota_j) \). In other words, the Goto function in the GIG case is a PDA and the set of viable prefixes with associated indices is a context-free language.

We will describe the Goto function adding the possibility to compute the possible stack values of the PDA for each state. The GIG function Goto has three parameters a) \( IT \) is a set of items b) \( X \) is a grammar symbol and c) \( IN \) is as e to fp a i a r s \((i, s)\) where \( i \) is an index symbol and \( s \) is a state defined by a set closure. We will say an index \( i \) is in \( IN \) if \( (i, s) \) is in \( IN \). The operation \( \text{goto}(IT, X, IN) \) is defined to be the pair \((IT_2, IN_2)\) where either 1 or 2 applies:

1. \( IT_2 \) is the closure of the set of all items \( [A \rightarrow \alpha X \cdot \beta] \) such that \( [A \rightarrow \alpha \cdot X \beta] \) is in \( IT \)

\( and \) \( IN_2 \) is defined to be the set of indices with an associated state such that:

- **Case A:** \( \alpha \) is \( \epsilon \) and \( X \) is a terminal \( a \) (i.e. \( [A \rightarrow \alpha \cdot X \beta] \) is in \( IT_2 \)):
  - If \( \mu \) is \( \delta \) in \( I \) then \( (\delta, IT_2) \) is in \( IN_2 \) and \( IN \) is in \( \text{predecessor}(\delta, IT_2) \).
  - If \( \mu \) is \( \epsilon \) then \( IN \) is in \( IN_2 \)
  - If \( \mu \) is \( \delta \) and \( \delta \) is in \( IN \) then every \( (\delta, IT) \) in \( IN \) is in \( IN_2 \)
  - If \( \mu \) is \( \delta \) and \( \delta \) is in \( IN \) then \( \text{predecessor}(\delta, IT) \) is in \( IN_2 \).

- **Case B:** If \( X \) is a non-terminal \( B \) then \( IN_3 \) is in \( IN_2 \) such that \( [B \rightarrow \gamma \cdot \] \) is in \( (IT_3, IN_3) \).

- In other words, the auxiliary stack has the configuration corresponding to the completed item B.

- **Case C:** If \( \alpha \) is not \( \epsilon \) \( X \) is a terminal \( a \) then \( IN = IN_2 \) (i.e. \( [A \rightarrow \alpha \cdot X \beta] \) is in \( IT_2 \) )

When the second parameter of the Goto function is \( \epsilon \) then \( \text{goto}(IT, \epsilon, IN) \) is defined to be:

2. \( IT_2 \) is the closure of the set of all items

- \( [B \rightarrow \delta \cdot \beta] \) such that \( [A \rightarrow \alpha \cdot B \beta] \) is in \( IT \) and \( \delta \) is in \( IN \) and \( IN_2 \) is defined such that \( \text{predecessor}(\delta, IT) \) is in \( IN_2 \) for every \( (\delta, IT) \) in \( IN \)

- \( [B \rightarrow \beta \cdot \beta] \) such that \( [A \rightarrow \alpha \cdot B \beta] \) is in \( IT \) and \( \delta \) is in \( IN \) and \( IN_2 \) is defined such that every \( (\delta, IT) \) in \( IN \) is in \( IN_2 \)

Consequently the Goto function for the grammar \( G_{wcc} \) is depicted in figure 2, with a PDA, where the first element of the pair is the symbol \( X \) and the second indicates the operation on the PDA stack (e.g.: \( i \) indicates a push and \( \bar{i} \) indicates a pop and \( [\#] \) indicates a transition that requires an empty stack).
The construction of the collection of sets of items for a GIG grammar follows the algorithm used for context free grammars (e.g. [2]). The only changes are those made to the \textit{Closure} and \textit{Goto} operations we described above. The construction of an LR parsing table is shown in the following algorithm. The action table has three parameters, state, input (or $\epsilon$) and auxiliary stack, the possible values are: a) pairs of shift action and operations on the auxiliary stack: \textit{push}, \textit{pop} or none; b) reduce action (in this case no operation on the auxiliary stack is performed (same value is kept). The \textit{Goto} table has the usual properties, and no operation on the auxiliary stack. The notation used is a follows: capital letters $(A, B)$ denote non-terminals lower case letters $(a)$ terminals, $\alpha$ is a non empty sequence of nonterminals and terminals and $\beta$ is a possible empty sequence of nonterminals and terminals, $\delta$ is an index and $\mu$ is either $\delta$, $\bar{\delta}$, $[\delta]$. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{transition_diagram.png}
\caption{Transition diagram for the PDA recognizing the set of valid prefixes, annotated with indices for the grammar $G_{wcw}$.}
\end{figure}
Algorithm 1 (Constructing an SLR Parsing Table)

Input. An augmented GIG grammar $G'$.
Output. The SLR parsing table functions action and goto

1. Construct $C = \{I_0, I_1, ..., I_n\}$, the collection of sets of LR(0) items for $G'$.
2. The goto transitions for state $i$ are constructed for all non-terminals $A$ using the rule: If $\text{goto}(i, A, \epsilon) = (I_j, I_{N_j})$, then set action$[i, A, \epsilon]$ to “reduce $A \rightarrow \mu"$ for all $\mu$ in $\text{FOLLOW}(A)$; where $A$ may not be $S'$.

3. If any conflicting actions are generated by the above rules, we say the grammars

4. the initial state of the parser is the one constructed from the set of items

The output transition table for the grammar $G_{wcw}$ is shown in figure 3. This table is used with an obvious extension of the context free algorithm, adding the additional stack to compute the corresponding actions depicted in the table.

Figure 4 shows the moves of the LR parser on the input string $abcab$.

5 Conclusions

We have made a short review of GIGs, GILs and their most important properties. We showed the descriptive power of GIGs is beyond CFGs. CFLs are properly included in GILs by definition. We showed also that GILs include some languages that are not in the LIL/TAL family nor in Weir’s control hierarchy. We presented the corresponding automata, its deterministic version and the algorithm to build LR parsing tables for deterministic GILs, the main contribution
of this paper. This approach can be extended to GLR parsing approaches using a graph-structured stack for the auxiliary stack. A similar approach was used in the Earley algorithm we presented in [6].

Acknowledgments

This work was partially supported by NLM Grant R01 LM06649-02.

References


Appendix

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(s1, p i) (s2, p j) (s3, _)</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>(s1, p i) (s2, p j) (s3, _)</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>(s1, p i) (s2, p j) (s3, _)</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>(s4, pop) (s5, pop) (s6, _)</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>(s4, pop) (s5, pop) (s6, _)</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>(s4, pop) (s5, pop) (s6, _)</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(s12, _)</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>(s13, _)</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>acc</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3. Action/Goto Table

<table>
<thead>
<tr>
<th>Stack 1</th>
<th>Stack2</th>
<th>Remaining input</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>#</td>
<td>Initial ID</td>
</tr>
<tr>
<td>2</td>
<td>0a1</td>
<td>#i</td>
<td>bcab$</td>
</tr>
<tr>
<td>3</td>
<td>0a1b2</td>
<td>#ij</td>
<td>cab$</td>
</tr>
<tr>
<td>4</td>
<td>0a1b2c3</td>
<td>#ij</td>
<td>ab$</td>
</tr>
<tr>
<td>5</td>
<td>0a1b2c35</td>
<td>#i</td>
<td>ab$</td>
</tr>
<tr>
<td>6</td>
<td>0a1b2c354</td>
<td>#i</td>
<td>ab$</td>
</tr>
<tr>
<td>7</td>
<td>0a1b2c3546</td>
<td>#</td>
<td>ab$</td>
</tr>
<tr>
<td>8</td>
<td>0a1b2c354R10</td>
<td>#</td>
<td>ab$</td>
</tr>
<tr>
<td>9</td>
<td>0a1b2c354R10a</td>
<td>#</td>
<td>b$</td>
</tr>
<tr>
<td>10</td>
<td>0a1b2c354R11</td>
<td>#</td>
<td>b$</td>
</tr>
<tr>
<td>11</td>
<td>0a1b2c354R11b</td>
<td>#</td>
<td>$</td>
</tr>
<tr>
<td>12</td>
<td>0a1b2c3R9</td>
<td>#</td>
<td>$</td>
</tr>
<tr>
<td>13</td>
<td>0a1b2S8</td>
<td>#</td>
<td>$</td>
</tr>
<tr>
<td>14</td>
<td>0a1S7</td>
<td>#</td>
<td>$</td>
</tr>
<tr>
<td>15</td>
<td>0S14</td>
<td>#</td>
<td>$</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>#</td>
<td>$</td>
</tr>
</tbody>
</table>

Fig. 4. Sequence of Moves of the DLR-2PDA on input abcab
Abstract. We study the disjunctive binary sequence introduced by Ehrenfeucht and Mycielski in [1]. The match length associated to the bits of the sequence is shown to be a crucial tool in the analysis of the sequence. We show that the match length between two consecutive bits in the sequence differs at most by 1 and give a lower bound for the limiting density of the sequence. Experimental computation in the automata package has been very helpful in developing these results.

1 The Ehrenfeucht-Mycielski Sequence

An infinite sequence is disjunctive if it contains all finite words as factors. In [1] Ehrenfeucht and Mycielski introduced a method of generating a disjunctive binary sequence based on avoiding repetitions. To construct the Ehrenfeucht-Mycielski (EM) sequence $U$, start with a single bit 0. Suppose the first $n$ bits $U_n = u_1 u_2 \ldots u_n$ have already been chosen. Find the longest suffix $v$ of $U_n$ that appears already in $U_{n-1}$. Find the last occurrence of $v$ in $U_{n-1}$, and let $b$ be the first bit following that occurrence of $v$. Lastly, set $u_{n+1} = \overline{b}$, the complement of $b$. It is understood that if there is no prior occurrence of any non-empty suffix the last bit in the sequence is flipped. The resulting sequence starts like so:

$$01001101011100010000111101100101001001110$$

see also sequence A038219 in Sloane’s catalog of integer sequences, [2]. The in the title of their paper the authors ask somewhat tongue-in-cheek how random their sequence is. As a first step towards understanding the properties of $U$ they show that $U$ is indeed disjunctive and conjecture that the limiting density of 1’s is $1/2$.

1.1 Preliminary Data

To get a better understanding of $U$ it is natural to generate a few thousand bits of the EM sequence using standard string matching algorithms. In a high-level environment such as Mathematica, see [3], a few lines of code suffice for this. In our work we use an automata theory package built on top of Mathematica that provides a number of tools that are helpful in the analysis of $U$, see [4].
Fig. 1. The first $2^{12}$ bits of the Ehrenfeucht-Mycielski sequence

for a recent description of the package. The first $2^{12}$ bits, in row-major order, are shown in figure [1] The pattern seems surprisingly indistinguishable from a random pattern given the simplicity of the definition of the sequence.

More interesting is a plot of the census function for $U$: nearly all words of length $k$ appear already among the first $2^k$ bits of the sequence. Thus, an initial segment of the EM sequence behaves almost like a de Bruijn sequence, see [5]. Define the cover $\text{cov}(W)$ of a word $W$, finite or infinite, to be the set of all its finite factors, and $\text{cov}_k(W) = 2^k \cap \text{cov}(W)$. Here we write $2$ for the two-symbol alphabet $\{0, 1\}$. The census function $C_k(n) = |\text{cov}_k(U_n)|$ for the EM sequence increases initially at a rate of 1, and, after a short transition period, becomes constant at value $2^k$. In figure [2] the green line stands for $k = 9$, blue for $k = 10$, and red for $k = 11$.

Another surprising picture emerges when one considers the length of the longest suffix $v$ of $U_n = u_1 u_2 \ldots u_n$ that matches with a previous occurrence. We write $\mu(n)$ for the suffix, and $\lambda(n) = |\mu(n)|$ for its length. As with the census function, the match length function $\lambda$ increases in a very regular fashion. Indeed, in most places the length of the match at position $n$ is $\lfloor \log_2 n \rfloor$. To visualize $\lambda$ it is best to collapse runs of matches of the same length into a single data point. The plot [3] uses the first $2^{15}$ bits of the sequence. It is immediate from the definitions that the match length can never increase by more that 1 in a single step. The plot suggests that the match lengths also never drop by more than 1 in a single step, a fact that will be established below. The data also suggest that the match length function is nearly monotonic: once the first match of length $k$ has occurred, all future matches are of length at least $k - 2$. If true, this property would imply balance of the EM sequence, see section [4].
Fig. 2. The census function for the Ehrenfeucht-Mycielski sequence for words of lengths $k = 9, 10, 11$

### 1.2 Generating Long Initial Segments

Clearly it would be helpful to test whether the patterns observed in the first few thousands of bits extend to longer initial segments, say, the first few million bits. To generate a million bits one has to resort to faster special purpose algorithms. As far as the complexity of $U$ is concerned, it is clear that the language $\text{pref}(U)$ of all prefixes of $U$ fails to be regular. Hence it follows from the gap theorem in [6] that $\text{pref}(U)$ cannot be context-free. The obvious practical approach is to use a variant of the KMP algorithm. Suppose $k$ was the length of the previous match. We can scan $U_n$ backwards and mark the positions of the nearest matches of length $k - 2, k - 1, k, k + 1$. If no such match appears we have to revise the near-monotonicity conjecture from above. Of course, the scan can be terminated immediately if a match of length $k + 1$ appears. If one implements this algorithm in an efficient language such as C++ it is straightforward to generate a few million bits of $U$.

Much better results can be achieved if one abandons pattern matching entirely and uses an indexing algorithm instead. In essence, it suffices to maintain, for each finite word $w$ of some fixed length at most $k$, the position of the last occurrence of that word in the prefix so far constructed. This is done in brute-force tables and quite straightforward except at places where the match length function assumes a new maximum. A detailed description of the algorithm can be found in [7]. The reference shows that under the assumption of near-monotonicity discussed in section 1.3 one can generate a bit of the sequence in amortized constant time. Moreover, only linear space is required to construct an initial segment of the sequence, so that a simple laptop computer suffices to generate the first billion bits of the sequence in less than an hour.

As far as importing the bits into automata there are two choices. Either one can read the precomputed information from a file. Note, though, that storing
the first billion bits in the obvious bit-packed format requires 125 million bytes, and there is little hope to decrease this amount of space using data compression: the very definition of the EM sequence foils standard algorithms. For example, the Lempel-Ziv-Welch based gzip algorithm produces a “compressed” file of size 159,410 bytes from the first million bits of the EM sequence. The Burrows-Wheeler type bzip2 algorithm even produces a file of size 165,362 bytes.

The other options exploits the fact that Mathematica offers a communication protocol that allows one to call external programs directly from the kernel. This feature is used in automata extensively to speed up crucial algorithms.

1.3 Assorted Conjectures

It is clear from data plots as in the last section that the EM sequence has rather strong regularity properties and is indeed far from random. In their paper [1] Ehrenfeucht and Mycielski ask if their sequence is balanced in the sense that the limiting frequency of 0’s and 1’s is 1/2. More precisely, for any non-empty word $x \in 2^*$ let $\#_1 x$ be the number of 1’s in $x$. Define the density of $x$ to be $\Delta(x) = \frac{\#_1 x}{|x|}$. The following conjecture is from [1]:

Conjecture 1. Balance

In the limit, the density of $U_n$ is 1/2.

Convergence seems to be very rapid. E.g., $\Delta(U_{2000000}) = 1000195/2000000 = 0.5000975$. It is shown in [8] that the density is bounded away from 0, and the argument given below provides a slightly better bound, but the balance conjecture remains open. To show balance, it suffices to establish the following property of the match length function.
Conjecture 2. Near Monotonicity

Any match of length $k$ is followed only by matches of length at least $k - 2$.

Near monotonicity implies rapid convergence of the density. We will prove a weaker a monotonicity property, namely that any match of length $k$ is followed only by matches of length at least $k/2$. This suffices to show that the limiting density is bounded away from 0. Another interesting property of $U$ is the rapid growth of the census function, simultaneously for all $k$.

Conjecture 3. Growth Rate

Any word of length $k$ appears in the first $O(2^k)$ bits of the sequence.

As a matter of fact, a bound of $2^{k+2}$ appears to suffice, but it is unclear what the growth rate of the number of words that fail to appear already at time $2^{k+1}$ is. We originally conjectured a bound of $2^{k+1}$ but had to revise it after Hodsdon computed the first billion bits of the sequence, see [7]. The last two conjectures hold true for the first billion bits of the sequence.

We note in passing another apparent structural property that becomes visible from the data. The plot of the match lengths suggests that they grow in a very regular fashion. It is natural to inquire about the position of the match in $U_n$, i.e., the position of the nearest occurrence of the suffix $v$ in $U_n$ associated with the next bit. Figure 4 shows the positions of the first $2^{14}$ matches. The available range of positions for the matches forms a staircase, with a few outliers, and the match positions essentially form square blocks of size $2^k$. The outliers are due to the internal dynamics of the sequence, see section 2.2 below, but match positions are very poorly understood at present.

![Fig. 4. Match positions in the first $2^{14}$ bits of the Ehrenfeucht-Mycielski sequence](image-url)
2 Recurrence and the Internal Clock

With a view towards computational support, it is convenient to think of the EM sequence as tracing a path in a de Bruijn $B_k$. We write $B_k(n)$ for the subgraph of $B_k$ induced by the edges that lie on the path traced by $U_n$. Likewise, $\overline{B}_k(n)$ denotes the complement of $B_k(n)$, i.e., the subgraph obtained by removing all the edges that lie on the path traced by $U_n$. We also assume that isolated vertices are removed. It is easy in automata to generate and inspect these graphs for a reasonably wide range of parameters. This type of experimental computation turned out to be very helpful in the discovery of some of the results in the next section, and in avoiding dead-ends in the development of some of the proofs.

As a first step towards the analysis of the dynamics of $U$, from the definition of $U$ we have the following fact.

**Proposition 1.** Alternation Principle

*If a vertex $u$ in $B_k(n)$ appears twice in $U_{n-1}$ it has out-degree 2.*

As we will see, the condition for alternation is very nearly the same as having in-degree 2. It is often useful to consider the nodes in $B_k$ that involve a subword $v$ of length $k-1$. Clearly, there are exactly four such nodes, and they are connected by an alternating path of the form:

$$a \ x \ v \ b \ \pi v \ a$$

We will refer to this subgraph as the zigzag of $v$. Since $B_k$ is the line graph of $B_{k-1}$, the zigzag of $v$ corresponds to the node $v$ and its 4 incident edges in $B_{k-1}$. It follows from the last proposition that the path $U$ can not touch a zigzag arbitrarily.

**Proposition 2.** No Merge Principle

*The path $U$ can not touch a zigzag in exactly two edges with the same target.*

In particular $v$ is a match if, and only if, all the nodes in the zigzag of $v$ have been touched by $U$.

2.1 The Second Coming

Since we are dealing with a binary sequence one might suspect the initial segments $U_{2^k}$ to be of particular interest, a suspicion borne out by figures 2 and 4. However, it turns out that there are other, natural stages in the construction of the EM sequence associated with the first repetition of the initial segments of the sequence. They determine the point where the census function first deviates from linear growth. First, a simple observation concerning the impossibility of repeated matches. Note that the claim made here is easy to verify using some of the graph algorithms in automata.

**Proposition 3.** Some initial segment $U_n$ of $U$ traces a simple cycle in $B_k$, anchored at vertex $U_k$. Correspondingly, the first match of length $k$ is $U_k$. 
Proof. Since \( U \) is infinite, it must touch some vertex in \( B_k \) twice. But by proposition 2, the first such vertex can only be \( U_k \), the starting point of the cycle. □

The proposition suggests to define \( \Lambda(t) = \max(\lambda(s) \mid s \leq t) \) to be the length of the longest match up to time \( t \). Thus, \( \Lambda \) is monotonically increasing and changes value only at the second occurrence of an initial segment. We write \( \tau_k \) for the time when \( U_k \) is encountered for the second time. Note that we have the upper bound \( \tau_k \leq 2^k + k - 1 \) since the longest simple cycle in \( B_k \) has length \( 2^k \).

The fact that initial segments repeat provides an alternative proof of the fact that \( U \) is disjunctive, see [1] for the original argument.

**Lemma 1.** The Ehrenfeucht-Mycielski sequence \( U \) is disjunctive.

**Proof.** It follows from the last proposition that every factor of \( U \) occurs again in \( U \). Now choose \( n \) sufficiently large so that \( H = B_k(n) = B_k(m) \) for all \( m \geq n \). Since every point in \( H \) is touched by \( U \) at least twice, it must have out-degree 2 by alternation. But the only such graph is \( B_k \) itself. □

It follows that every word appears infinitely often on \( U \), and we can define \( \tau_i^w, i \geq 0, \) to be the position of the \( i \)th occurrence of word \( w \) in \( U \). As always, this is interpreted to mean the position of the last bit of \( w \). Define \( \tau_i^U \) to be \( \tau_i^{U_k} \), so \( \tau_0^k = k \) and \( \tau_1^k = \tau_k \). Also note that \( \tau_{k+1} = \tau_{2^k} + 1 \).

**Proposition 4.** Any word of length \( k \) other than \( U_k \) appears exactly once as a match. The initial segment \( U_k \) appears exactly twice. Hence, the total number of matches of length \( k \) is \( 2^k + 1 \).

**Proof.** First suppose \( u \in 2^k \) is not an initial segment of \( U \). By lemma 1 \( a u \) and \( \overline{a} u \) both appear in \( U \). The first such occurrences will have \( u \) as match. Clearly, from then on \( u \) cannot appear again as a match. Likewise, by 1 any initial segment \( u = U_k \) must occur twice as a match since there are occurrences \( u, a u \) and \( \overline{a} u \). As before, \( u \) cannot reappear as a match later on in the sequence. □

### 2.2 Rounds and Irregular Words

Proposition 3 suggests that the construction of \( U \) can be naturally decomposed into a sequence of rounds during which \( \Lambda \) remains constant. We will refer to the interval \( R_k = [\tau_k, \tau_{k+1} - 1] \) as the \( k \) principal round. During \( R_k \), the maximum match function \( \Lambda \) is equal to \( k \), but \( \lambda \) may well drop below \( k \). Up to time \( t = \tau_{k+1} - 1 \) the EM sequence traces two cycles \( C_0 \) and \( C_1 \) in \( B_k \), both anchored at \( u = U_k \). \( C_0 \) is a simple cycle, and the two cycles are edge-disjoint. Note that the complement \( B_k(t) = B_k - C_0 - C_1 \) consists only of degree 2 and, possibly, degree 4 points, the latter corresponding to words of length \( k \) not yet encountered at time \( t \). The strongly connected components are thus all Eulerian.

![Diagram](image-url)
When $U$ later touches one of these components at $u_0$, by necessity a degree 2 point, we have the following situation: $v = aw$ and $u_0 = wb$ so that the sequence look like $...awb...awb...$. Thus, the first two occurrences of $w$ are preceded by the same bit. Such words will be called irregular and we will see shortly that the first three occurrences of any irregular word are of the form $...awb...awb...$. For the sake of completeness, we distinguish between irregular, regular and initial words. It is easy to see that all words $0^k$ and $1^k$, $k \geq 2$ are irregular. There seem to be few irregular words; for example, there are only 12 irregular words of length 10. It is clear from the definitions that whenever $v$ occurs as a match, all its prefixes must already have occurred as matches. Because of irregular words, the situation for suffixes is slightly more complicated, but we will see that they too occur as matches with a slight delay.

Our interest in irregular words stems from the fact that they are closely connected with changes in match length. Within any principal round, $\lambda$ can decrease only when an irregular word is encountered for the second time, and will then correspondingly increase when the same word is encountered for the third time, at which point it appears as a match. First, increases in match length.

**Lemma 2.** Suppose the match length increases at time $t$, i.e., $\lambda(t+1) = \lambda(t) + 1$, but $\Lambda$ does not increase at time $t$. Then $v = \mu(t)$ is irregular and $t = \tau_v^w$. Moreover, at time $s = \tau_v^w$ the match length decreases: $\lambda(s) > \lambda(s + 1)$.

**Proof.** Set $k = |v|$ and consider the edges incident upon $v$ in $B_k$ at time $t$. The dashed edge indicates the last step.

Since the match length increases, both edges $(v, wb)$ and $(v, w\overline{b})$ must already lie on $U_t$. But that means that the edge $(au, v)$ must appear at least twice on $U_t$, and $v$ is irregular. Now consider the time $s = \tau_v^w$ of the second appearance. We must have $s > r = \tau_k^v$. But the strongly connected component of $v$ in the residual graph $B_k(r)$ consists only of degree 2 and, possibly, degree 4 points; point $v$ itself is in particular degree 2. As a consequence, $U$ must then trace a closed path in this component that ends at $v$ at time $t = \tau_v^w$. Lastly, the match length at time $s + 1$ is $k$, but must have been larger than $k$ at time $s$. \[\square\]

Thus all changes in match length inside of a principal round are associated with irregular words. The lemma suggests the following definition. A minor round (of order $k$) is a pair $(r, s)$ of natural numbers, $r \leq s$, with the property that $\lambda(r - 1) \geq k + 1, \lambda(t) \leq k$ for all $t, r \leq t \leq s$, and $\lambda(s + 1) \geq k + 1$. Since trivially $\lambda(t + 1) \leq \lambda(t) + 1$, the last condition is equivalent to $\lambda(s + 1) = k + 1$.

Note that minor rounds are either disjoint or nested. Moreover, any minor round that starts during a principal round must be contained in that principal
round. We can now show that match length never drops by more than 1 at a time.

**Lemma 3.** Let \((r, s)\) be a minor round. Then \(\lambda(r - 1) = \lambda(r) + 1 = \lambda(s + 1)\).

**Proof.** From the definition, for any minor round \((r, s)\) we have \(\lambda(s + 1) - \lambda(r - 1) \leq 0\). Now consider the principal round for \(k\). As we have seen, all minor rounds starting before \(R_k\) are already finished at time \(\tau_k^1\). But if any of the minor rounds during the \(k\) principal round had \(\lambda(s + 1) - \lambda(r - 1) < 0\) then the match length at the end of \(R_k\) would be less than \(k\), contradicting the fact that the match length increases to \(k + 1\) at the beginning of the next principal round. \(\Box\)

Hence, there cannot be gaps between two consecutive match length values.

**Theorem 1. No-Gap**
For all \(n\), \(\lambda(n) - 1 \leq \lambda(n + 1) \leq \lambda(n) + 1\).

### 2.3 A Lower Bound

It follows from the last section that for \(u\) not an initial segment, \(\tau_1^u \in R_k\) implies that \(u\) matches at some time \(t \in R_k\). We will say that \(u\) **matches with delay** at time \(\tau_1^u\).

**Lemma 4.** Let \(u\) be a word, not an initial segment. At time \(\tau_3^u\) both \(0u\) and \(1u\) match with delay.

**Proof.** First suppose that \(u\) is regular. Consider the neighborhood of \(u\) in \(B_k\) where \(k = |u|\). In the following figure, the edge labels indicate one way \(U\) may have passed through \(u\) by time \(\tau_3^u\). Note that our claim follows trivially if both \(au\) and \(au\) appear twice on \(U_{\tau_3^u}\), so we only need to deal with the asymmetric case.

![Diagram](Diagram.png)

Since \(ub\) appears twice, it must match, with delay. But then both \(\overline{au}b\) and \(au\overline{b}\) must appear, so \(\overline{au}\) appears twice and must match, with delay. A similar argument covers the remaining case. For \(u\) irregular the second encounter entails a third as indicated in the following figure. It suffices to deal with a fourth hit as indicated below.

![Diagram](Diagram.png)

But then \(ubc\) is also irregular, and we must have an occurrence of \(\overline{abc}\), with delay. \(\Box\)
Lemma 5. If \( uab \) has matched at time \( t \), then both \( 0u \) and \( 1u \) match at time \( t \), with delay.

Proof. From the last lemma, our claim is obvious as long as \( u \) is not an initial segment. So suppose \( u = U_k \) and consider the first 5 occurrences of \( u \):

\[ uab \ldots xua \ldots \bar{x}ua \ldots xua b \ldots xua bc \]

Note that the second occurrence of \( xuab \) is before the end of round \( R_{k+2} \), so both \( xu \) and \( xu \) must have matched before the end of that round. \( \square \)

Corollary 1. If a word \( u \) of length \( k \) matches at time \( t \), then all words of length at most \( \lfloor k/2 \rfloor \) have matched at time \( t \), with delay.

From the corollary we obtain the lower bound \( \tau_k = \Omega(\sqrt{2}^k) \). It follows from an argument in [8] that this yields a lower bound of \( 0.11 \) for the asymptotic density of \( U \), a far cry from the observed value of \( 1/2 \).

3 Density and Near Monotonicity

The density of a set \( W \subseteq 2^k \) is defined by \( \Delta(W) = \frac{1}{|W|} \sum_{x \in W} \Delta(x) \). To keep notation simple, we adopt the convention that a less-than or less-than-or-equal sign in an expression indicates summation or union. E.g., we write \( \sum_{0 \leq i < p} \binom{k}{i} \) for \( \sum \) all words containing exactly \( p \) many 1’s. Thus, \( |2^{k,p}| = \binom{k}{p} \). Clearly \( \Delta(2^k) = 1/2 \) by symmetry. A simple computation shows that, perhaps somewhat counterintuitively, \( \Delta(2^{k, \leq k/2}) = 1/2 \) for all \( 1/2 \leq \varepsilon \leq 1 \).

Now suppose \( W \subseteq 2^k \) is a set of cardinality \( m \). What is the least possible density of \( W \)? Clearly, a minimal density set \( W \) must have to form \( 2^{k, \leq p} \cup A \) where \( A \subseteq 2^{k, p+1} \). If \( m \) forces \( p \geq k/2 \), then asymptotically the density of \( W \) is \( 1/2 \). Indeed, we will see that \( m = \Omega(2^k) \) suffices. Let \( 0 \leq p \leq k \). From the definition of density we have

\[
\Delta(2^{k, \leq p}) = \sum_{0 \leq i \leq p} \binom{k}{i} i/k = 1/2 - \left( \frac{\sum \binom{k-1}{i} i/k}{\binom{k}{p}} \right) + 2^{-1}
\]

Let \( p = |\varepsilon k| + c \) where \( c \in \mathbb{Z} \) is constant. As long as \( 1/2 \leq \varepsilon \leq 1 \) we obtain density \( 1/2 \) in the limit. However, this is far as one can go.

Lemma 6. Let \( 0 \leq \varepsilon < 1/2 \) and \( p = |\varepsilon k| + c \) where \( c \in \mathbb{Z} \) is constant. Then

\[ \lim_{k \to \infty} \binom{k}{p} \varepsilon / (1 - 2\varepsilon) \]

Proof. For the sake of brevity we write \( \gamma = \binom{k}{p} \). First note that the density of \( 2^{k, \leq \varepsilon k} \) is clearly bounded from above by \( \varepsilon \). Since \( \Delta(2^{k, \leq \varepsilon k}) = \frac{7}{2^{\gamma+1}} \) it follows
that $\gamma \leq \frac{1}{1-2\varepsilon}$. For the opposite direction we rewrite the individual quotients of binomial coefficients in terms of Pochhammer symbols as $\left(\frac{k}{p-i}\right)/\left(\frac{1}{p}\right) = \frac{(p-i+1)}{(k-p+1)}$.

Hence the limit of $\left(\frac{k}{p-i}\right)/\left(\frac{1}{p}\right)$ as $k$ goes to infinity is $\left(\frac{1}{1+\varepsilon}\right)^i$. Now consider a partial sum $\sum_{i=1}^{n} \left(\frac{k}{p-i}\right)/\left(\frac{1}{p}\right) \leq \gamma$ where $n$ is fixed. Then

$$\sum_{i=1}^{n} \left(\frac{k}{p-i}\right)/\left(\frac{1}{p}\right) \rightarrow \sum_{i=1}^{n} \left(\frac{1}{1+\varepsilon}\right)^i = \frac{1}{1-2\varepsilon} \left(1 - \left(\frac{1}{1-\varepsilon}\right)^n\right)$$

as $k$ goes to infinity. But then $\lim_{k \to \infty} \gamma \geq \frac{\varepsilon}{1-2\varepsilon}$. Thus, in the limit $\gamma = \frac{\varepsilon}{1-2\varepsilon}$. □

**Corollary 2.** Let $0 \leq \delta \leq 1/2$. Then $\lim_{k \to \infty} \Delta(2^k; \leq \delta k) = \delta$.

The definition of density extends naturally to multisets $A, B \subseteq 2^k$ via $\Delta(A + B) = \frac{|A\Delta(A) + B\Delta(B)|}{|A + B|}$. Assuming near monotonicity, we can now establish balance of $U$ by calculating the limiting density at times $\tau_k$. Let us say that $\lambda$ is $c$-monotonic if $\forall t, s \lambda(t + s) \geq \lambda(t) - c$. Thus, it seems that $\lambda$ is 2-monotonic, but the argument below works for any constant $c$.

**Theorem 2.** If $\lambda$ is $c$-monotonic for some constant $c$, then the Ehrenfeucht-Mycielski sequence is balanced.

**Proof.** Assume otherwise; by symmetry we only have to consider the case where for infinitely many $t$ we have $\Delta(U_t) < \delta_0 < 1/2$. Let $\tau_{k+c} \leq t < \tau_{k+c+1}$ and consider the multiset $W = \text{cov}_t(U_t)$. For $t$ sufficiently large $\Delta(W) < \delta_0$. Since all matches after $t$ have length at least $k$ by our assumption, certainly $2^k \subseteq W$. Since all words of length $k + c + 1$ on $U_t$ are unique, there is a constant bounding the multiplicities of $x \in 2^k$ in $W$ and we can write $W = 2^k + V$ where $\forall x \in 2^k (V(x) \leq d)$. Let $\delta = \Delta(V)$ and $m = |V|$, so that

$$\delta_0 < \Delta(W) = \frac{2^k/2 + m \cdot \delta}{2^k + m}.$$ 

It follows that $2^k(1 - 2\delta_0) \leq m(\delta_0 - \delta) \leq m$ so that $m = \Omega(2^k)$.

On the other hand, we must have $\delta_0 \geq \Delta(V) \geq \Delta(d \cdot 2^k; \leq p) = \Delta(2^k; \leq p)$. To see this, note that if for some $x \in 2^k$, $q/k = \Delta(x) < \Delta(2^k + d \cdot 2^k; \leq q)$ then $2^k + d \cdot 2^k; \leq q$ minimizes the density of all multisets with multiplicities bounded by $d$ that include $x$. From the last corollary we get $p \leq \delta_0 k$. Using Sterling approximation we see that the cardinality $m$ is bounded by $d \left(\begin{array}{l}k \\ \leq \delta_0 k \end{array}\right) \leq d + d\delta_0 k \left(\begin{array}{l}k \\ \delta_0 k \end{array}\right) \approx d + d \sqrt{\frac{\delta_0 k}{2\pi (1-\delta_0)}},$ where $H(x) = -x \log x - (1-x) \log(1-x)$ is the binary entropy function over the interval $[0,1]$. It is well-known that $H$ is symmetric about $x = 1/2$ and concave, with maximum $H(1/2) = 1$. Hence $2^{H(\delta_0)} < 2$, contradicting our previous lower bound. Hence, the density of $W$ approaches $1/2$, as required. □
4 Conclusion

We have established some regularity properties of the Ehrenfeucht-Mycielski sequence, notably the No-Gap conjecture and a weaker form of Near Monotonicity. A better analysis of the match length function should show that $\lambda$ is in fact 2-monotonic. Specifically, a study of the de Bruijn graphs $B_k$ in automata indicates that the strongly connected component of this graph has special properties that could be exploited to establish this claim. Alas, we are currently unable to give a complete proof. The construction of the Ehrenfeucht-Mycielski sequence easily generalizes to arbitrary prefixes: start with a word $w$, and then attach new bits at the end according to the same rules as for the standard sequence. It seems that all results and conjectures here seem to carry over, mutatis mutandis, to these generalized Ehrenfeucht-Mycielski sequences. In particular, they all appear to have limiting density $1/2$.

Source code and Mathematica notebooks used in the writing of this paper can be found at www.cs.cmu/~sutner.

References


The Longest Common Subsequence Problem  
A Finite Automata Approach *

Bořivoj Melichar and Tomáš Polcar

Department of Computer Science and Engineering  
Faculty of Electrical Engineering  
Czech Technical University  
Karlovo nám. 13, 121 35 Prague 2, Czech Republic  
{melichar,polcart}@fel.cvut.cz

Abstract. A new algorithm that creates a common subsequence automaton for a set of strings is presented. Moreover, it is shown that a longest common subsequence of two strings over a constant alphabet can be found in $O(|A|(|S_1| + |S_2| + \sum_{a \in A} |S_1|_a |S_2|_a))$ time, where $|A|$ is the size of the alphabet, $|S_i|$ is the length of the input string $i$, and $|S_i|_a$ is the number of occurrences of $a \in A$ in $S_i$.

The longest common subsequence problem is one of many unsolved problems in the computer science. This problem has an application in many areas as computational biology, computer-assisted music analysis, data compression, text analysis, and others.

A subsequence of given string $S$ is any string created from $S$ by deleting zero or more symbols of it. For the set of strings $P$, the common subsequence of $P$ is such string that is a subsequence of every string $S \in P$. The longest common subsequence is a common subsequence having maximal length.

The problem of finding the longest common subsequence of pair of strings and multiple strings has been studied almost over three decades. After the pioneering works of Hirschberg [1] and Hunt and Szymanski [2] many papers and reports has been published about the longest common subsequence of two strings. Most of them are based on the dynamic programming, which leads to the quadratic time and space complexity. The improvement was leading to the linear space complexity, while the time complexity remains still quadratic.

Hébrard and Crochemore [3] mentioned a finite automaton accepting all subsequences of a given string. Its left to right construction has been described in [4]. This finite automaton, in some sources called direct acyclic subsequence graph (DASG), will be referred here as subsequence automaton (SA). After that Crochemore and Tronček presented a position points based construction of common subsequence automaton (CSA) [5].

* This research has been partially supported by the Ministry of Education, Youth, and Sports of the Czech Republic under research program No. J04/98:212300014 (Research in the area of information technologies and communications) and by Grant Agency of Czech Republic grant No. 201/01/1433.
New automata approach based construction of CSA is presented in this paper and better upper bound of the time complexity is proven.

Let us denote $\text{Sub}(S)$ the set of all subsequences of a string $S$. To find the longest common subsequence of a set of strings $P = \{S_1, S_2, \ldots, S_m\}$, $2 \leq m$, the following three steps are used:

1. Construction of subsequence automaton $\text{SA}(S_i)$, $i = 1, 2, \ldots, m$, accepting $\text{Sub}(S_i)$ for each string $S_i \in P$.
2. Construction of common subsequence automaton $\text{CSA}(P)$ accepting the intersection of all sets $\text{Sub}(S_i)$: $\text{CSA}(P) = \text{Sub}(S_1) \cap \text{Sub}(S_2) \cap \ldots \cap \text{Sub}(S_m)$.
3. Finding the $\text{LCS}(P)$ using an algorithm for topological ordering well known from graph theory.

It is easy to see, that for all symbols $a$ longest common subsequence contains at most as much symbols $a$ as $\min \{|S_i|_a | i = 1, \ldots, m\}$. Thus, the length of the longest common subsequence is at most $\sum_{a \in A} \min \{|S_i|_a | 1 \leq i \leq m\}$.

Let $\mathcal{M} = (Q, A, q_0, \delta, F)$ be a finite automaton. Let we define for arbitrary $a \in A$ the set $Q(a) \subseteq Q$ as follows: $Q(a) = \{q | q \in \delta(p, a), a \in A, p, q \in Q\}$. The finite automaton is called homogenous finite automaton, if for all pairs of symbols $a, b \in A$, $a \neq b$ it holds $Q(a) \cap Q(b) = \emptyset$.

The left to right construction of subsequence automaton for string $S$ requires $O(|A| \cdot |S|)$ time. Therefore, the first step of presented algorithm can be done in $O(\sum_{i=1}^m |A| \cdot |S|)$ time. Moreover, such way created automaton is a homogenous finite automaton with the number of states $|Q| = |S| + 1$ and $|Q(a)| = |S|_a$.

As far as the common subsequence automaton is created as an intersection of subsequence automata (it accepts the intersection of languages accepted by subsequence automata), its set of states is a subset of cartesian product of states of subsequence automata. Since an intersection of homogenous automata results to a homogenous automaton, common subsequence automaton is homogenous too. Thus, the estimation of number of states of common subsequence automaton is at most $|Q| = \sum_{a \in A} \prod_{i=1}^m |S_i|_a + 1$. Therefore, the second step takes $O(m|A| \sum_{a \in A} \prod_{i=1}^m |S_i|_a)$ time. The time complexity of the third step is the same as the time complexity of second one. Thus, the time complexity of presented algorithm is $O(|A| (\sum_{i=1}^m |S_i| + m \sum_{a \in A} \prod_{i=1}^m |S_i|_a))$. Especially for two strings it is $O(|A| (|S_1| + |S_2| + \sum_{a \in A} |S_1|_a |S_2|_a))$.

References

AVA: An Applet for Visualizing FRACTRAN and Other Automata

Nathan Preston and Mukkai S. Krishnamoorthy
Rensselaer Polytechnic Institute
Troy NY 12180

Abstract. In this paper we present the design and features of the Automaton Visualization Applet (AVA). AVA is a tool for creating and editing graphs and automata which provides a friendly user interface. The applet supports finite automata, pushdown automata, Turing machines, and a model based on fractions called FRACTRAN, of which we give an overview. AVA allows the user to perform the actions of the various machines, e.g. testing input strings to see if they are accepted. The applet also supports importing and exporting graphs using eXtensible Graph Markup and Modeling Language (XGMML), an XML language for describing graphs.

1 Summary

AVA is a Java Applet which allows for the simple construction of graphs and automata, and also offers several embeddings for efficient layout and visualization. The available embeddings include Random, Stable, Force-Directed, Relaxed, Barycentric, and Level. The various computing models are implemented on top of the graph structure, which allows for consistency when creating, manipulating, and testing the machines. Because AVA is an applet, no installation is necessary, and it can be run on any Java-enabled web browser. AVA’s simple user interface allows for graphs to be created quickly and intuitively, and the visualization of several computational model offers educational benefits to those studying them.

AVA supports the following models of computation: finite automata, pushdown automata, Turing machines, and FRACTRAN model. The first three should be familiar to the reader; the latter we will consider shortly. Any of these machines can be created by the same method a graph is created, and then setting appropriate node and edge attributes. The user can easily test if an input string is accepted by the current machine, and can also generate a formal description of the current machine.

The FRACTRAN model is a universal computational machine, originally introduced by Conway[1]. A FRACTRAN "game" is played as follows:

- Start with a list of fractions $f_1, f_2, \ldots, f_k$ and a starting integer $N$.
- Multiply the current integer (initially $N$) by the first fraction in the list for which the product is an integer, to obtain a new integer.
Repeat the previous step until there is no such fraction for which the product above is integral. At this point the game stops. (Note that it is possible that the game will never stop.)

By considering the prime factors appearing in the various fractions as storage registers, we can manipulate these registers to perform computation.

**Fractran** games can easily be represented with graphs, where edges contain the appropriate fractions. For example, the graph in Figure 1 works as a simple adder. Similarly, we can create Fractran games that perform computations such as min, multiplication, prime numbers, and even the digits of π.

AVA uses a standardized intermediate format (XGMML) to represent graphs, making it compatible with other programs that use the same format. XGMML is an extension of GML (Graph Modelling Language), an XML standard for describing graphs. It provides a convenient tagged format for describing graphs, allowing different graphing applications to communicate. AVA can read XGMML descriptions of the various graphs/machines through the use of the SAXParser, and can also generate XGMML descriptions. The use of XGMML allows graphs and automata to be represented very accurately, as XGMML provides a means of representing both the structure of the graph and display information.

The reader is encouraged to visit the AVA web site, at http://www.cs.rpi.edu/projects/pb/edu. The web site includes source code, the user’s manual, XGMML examples, and an extended version of this paper [3].

**References**


Preliminary Experiments in Hardcoding Finite Automata

E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie

Department of Computer Science
University of Pretoria
Pretoria 0002, South Africa
{eketcha,bwatson,dkourie}@cs.up.ac.za

Abstract. Various experiments in hardcoding a single row of a transition table of a finite state machine directly into symbol-recognizing code are presented. Measurements are provided to show the time efficiency gains by various hardcoded versions over the traditional table-driven algorithm.

1 The Hardcoding Experiment

The most commonly used algorithm to determine whether an input string is in the language of a finite automaton (FA) assumes that the transition function of the FA is represented as a table in memory. However, if the same table is to be repeatedly used to recognize strings, the question arises: “Would it not be more efficient to use a variation of the algorithm that encapsulates the transition table information into the code itself – i.e. to rely on an algorithm that hardcodes the FA?”

We have started to explore this question by comparing the timing behavior of five hardcoded algorithms against the table-driven algorithm on a Linux platform. The table-driven algorithm and two hardcoded algorithms were written in C++ and compiled using the gnu C++ optimizing compiler. The remaining three hardcoded algorithms were written in NASM assembler. Given the current input symbol, the first C++ hardcoded algorithm relies on the switch statement to determine the next state. The second C++ hardcoded algorithm relies on a sequence of nested if-then-else statements to determine the next state from the input symbol. The first and second assembler algorithms mimic these high-level versions in assembler. The switch statement is mimicked at the assembler level by making use of a jump table. The nested if-then-else statements are mimicked by assembler code that essentially executes a linear search through the list of accepting symbols in the current state. Finally, the third assembler algorithm relies on a direct jump statement to execute a block of code whose location is computed from the current input symbol. All of these algorithms have been carefully designed to reflect the work to be done by some FA in accepting or rejecting a single character of some input string when the FA is in some arbitrary state. To provide code that would test whether a complete string is a member of the

© Springer-Verlag Berlin Heidelberg 2003
FAs language, one could simply extend any one of the algorithms in a relatively straightforward linear fashion.

An experiment was carried out in which the following steps were repeated several hundred times.

– Generate a row of a transition table such that the size and content of the set of accepting symbols are randomly determined; and each transition leads to some randomly determined next state. Regard the set of accepting symbols as the problem size.
– Generate the code that is associated with this transition row for each of the five hardcoded algorithms.
– Run this code as well as the relevant code for the table-driven algorithm, for each of the possible input symbols allowed by the alphabet under test. Each run is repeated twenty times. In each case the time taken to execute the code was recorded.

The resulting data revealed that the performance of each of the algorithms was substantially unrelated to the problem size. As a result, it seemed reasonable to base further comparisons of the six different coding possibilities on average values taken over all problem sizes. In each case, the jump table version is more than twice as fast as its nearest rival (the linear search version) and more than forty times faster than any of the high-level implementation versions, whether table-driven or hardcoded. Globally speaking, hardcoding implementations in assembler are faster than other methods. Moreover, hardcoding to a high-level language does not appear to be worthwhile, since the standard table-driven implementation seems to be slightly faster.

2 Conclusion

This study provides prima facie evidence that hardcoding in assembler could lead to significant performance improvements in relation to the table-driven algorithm. However, it would be naïve to assume that the recognition of an entire string merely involves a linear scale-up of the times obtained in these experiments. What needs to be explored is the paging, data caching and instruction caching impact on the table-driven approach (which uses a relatively small program but relies on a very large transition table as its data) and on the various hardcoded approaches (where the transition table is embedded into larger programs).
Computational Linguistic Motivations for a Finite-State Machine Hierarchy

Robert Kelly and Julie Carson-Berndsen

Department of Computer Science
University College Dublin

{robert.kelly,julie.berndsen}@ucd.ie

Abstract. Given the prominence of finite-state techniques in natural language processing, toolkits are required that can provide finite-state capability for applications in this domain. For example, work in the context of Multi-Agent Speech Recognition requires such finite-state capability. In particular, an interface is required both for processing finite-state representations and for automatically acquiring such representations. The finite-state representations are the traditional finite-state machines together with their stochastic equivalents. The representations are specified in XML which is chosen both for clarity and for reusability and portability across different platforms and applications. Motivated by these requirements this paper presents a Finite-State Machine Hierarchy. The hierarchy is an extendible object-oriented inheritance tree where each class in the hierarchy represents a particular finite-state machine. The hierarchy has been used to deliver the required interfaces with support for XML marked up machine structures.

1 Summary

A Finite-State Machine Hierarchy has been developed which collects together in a single structure multiple finite-state machine definitions and provides abstract algorithms for working with these machine definitions. The hierarchy supports finite-state machine definitions specified as XML, text or array structures.

An abstract FSM machine class resides at the apex of the hierarchy. This abstract class defines the algorithms required by all finite-state machines at an abstract level by incorporating abstract methods. By subclassing this abstract class, concrete machine classes inherit the algorithms and by implementing the abstract methods tailor them to their own idiosyncratic internal structure. In this way concrete machine classes are free to maintain different internal structures and still utilise the algorithms defined in the abstract machine class. This results in an extendible hierarchy where the process of defining a new type of concrete machine class, i.e. a machine class not part of the current hierarchy, is a simple one. The hierarchy is designed with the explicit intention of providing a programming interface for building applications that require finite-state components.
Fig. 1. The Finite-State Machine Hierarchy. A connecting line between two machine classes represents an inheritance relationship between those two classes.

The classes that are currently implemented as part of the hierarchy are illustrated in Figure 1. The internal structure of the abstract FSM class consists of a set of states with a single start state and a subset designated as final states and a set of transitions where the transition symbols can have an arbitrary number of tapes specified. The FSM class provides abstract operations to read from and write to intermediate finite-state structures in XML, text or array format. Also, an operation is defined allowing paths to be traced through a machine for a given sequence of input events to determine if the machine accepts the sequence as well-formed. Note that input events may specify multiple tapes and thus a sequence can be tested on multiple transition tapes of a machine. Through the use of abstract methods the FSM class captures the algorithmic similarity of the operations for all machine classes. The FST class maintains the same internal structure as the FSM class and tailors the operations defined in the FSM class to transducer structures. The SFST class has a variant internal structure of stochastic states and stochastic transitions and tailors the operations defined in the FSM class to stochastic transducer structures. The ALGERIAFST class maintains the same internal structure as the SFST class and provides additional operations that implement the ALGERIA machine induction algorithm [1]. This furnishes the ALGERIAFST class with the ability to induce its own internal structure of stochastic states and stochastic transitions from a set of training examples. The ALGERIAFST class can be used to induce multi-tape stochastic transducers with an arbitrary number of tapes and all induced transducer structures are necessarily deterministic.

The hierarchy has been used in the context of multi-agent speech recognition to learn descriptions of the phonotactics of languages and to process those descriptions [2]. For details on other practical applications of the hierarchy see [3, 4] for example.
References


The Effect of Rewriting Regular Expressions on Their Accepting Automata

Michiel Frishert\(^1\), Loek G. Cleophas\(^1\), and Bruce W. Watson\(^2\)

\(^1\) Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O.Box 513, NL-5600 MB Eindhoven, The Netherlands
m.frishert@stud.tue.nl, loek@loekcleophas.com

\(^2\) Department of Computer Science
University of Pretoria,
Pretoria 0002, South Africa
bruce@bruce-watson.com

Traditionally, regular expressions are transformed into (deterministic) finite automata, which are then used in, for example, searching and parsing. For an introduction to automata and regular expressions, refer to [1]. For every automaton, there is an infinite number of other automata that recognizes exactly the same language, although memory usage and performance can differ. This has lead to the wide-spread use of automata minimization techniques. In the same way, for every regular expression, there exists an infinite number of equivalent regular expressions, some of which lead to smaller/faster automata. This paper compares automaton sizes constructed from 10 different regular expressions, both before and after the rewriting transformation. Rewriting is done based on a rule set directly derived from the regular expression language (i.e. Kleene Algebra) axioms; these are listed in Table 1.

The first three automata tested are the Thompson NFA [2], the Position NFA [3], and the Follow ε-NFA [4]. The remaining three automata are the determinized counterparts of each NFA. For these DFA, the size of the minimized DFA is provided for comparison. For a list of the regular

Table 1. Rewrite Rules. Note that \(a, E, F\) are regular expressions

\[
\begin{array}{c|c|c|c|c}
\varepsilon \cdot E & \emptyset^* & \varepsilon & (E^*|F)^* & (E|F)^* \\
E \cdot \varepsilon & \emptyset^* & \varepsilon & (E)^* & E^* \\
E \cdot \emptyset & \emptyset^* & \varepsilon & (E)^* & E^* \\
a \cdot E|a \cdot F & a \cdot (E|F) & E^* & E^* & E^* \\
\emptyset \cdot E & \emptyset \cdot (E|F) & E^* & E^* & E^* \\
E \cdot \emptyset & E \cdot (E|F) & E^* & E^* & E^* \\
E | \emptyset & E & E & E^* & E^* \\
\varepsilon^* & \varepsilon & E \cdot E^* & E^* & E^* \\
\end{array}
\]

© Springer-Verlag Berlin Heidelberg 2003
expressions used in this test, see [5]. In Fig. 1 the results are presented visually. Clearly, rewriting has an impact on the automaton size. However, the impact varies between regular expressions and between automata.

Our first implementation reads its rules from a text file and is therefore quite slow: rewriting takes up to 10 times as long as automaton construction. However, once a set of good rules is selected, they can be hardcoded, thus leading to much better performance.

References


Building Context-Sensitive Parsers from CF Grammars with Regular Control Language

César Bravo* and João José Neto

Laboratório de Linguagens e Tecnologias Adaptativas
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, 158, tv. 3 – 05508-900 São Paulo, SP, Brasil
joao.jose@poli.usp.br
lupus@usp.br

Abstract. In this work we propose a method to derive Stack Automata [1] from context-free grammars with regular control languages [3]; by slightly restricting the operation of the machine we obtain a bottom-up parser which operates by reducing instances of the right-hand side of productions in a sentencial form to the corresponding left-hand side; since context-free grammars with regular control languages are Turing powerful and Stack Automata accept, at least, context-sensitive languages [2], the resulting devices are indeed context-sensitive parsers.

In this work, we propose a practical way to build parsers for context-sensitive languages following three steps:

1. Design a context-free grammar $G$ with regular control set $R$ for the target language,
2. Build the finite automaton corresponding to the regular expression $R$,
3. Obtain the corresponding one-way stack automaton introduced in this paper.

One-way stack automata, which are able to write anywhere onto the stack but only read once the input, are defined in [1], page 391, but in this paper their operation will be restricted by the following constraints:

1. Pre-processing: Push the whole input string onto the stack,
2. Processing: Nondeterministically scan the stack from top to bottom in order to determine some transition to be applied; if a match is found, then perform the transition. Afterwards, make the stack pointer indicate the top of the stack,
3. Halting Condition: Stop accepting the input just when the stack becomes empty.

Let $G = (N, T, P, S, R, F)$ be a context-free grammar with regular control language [3]. Note that the stack alphabet will be formed by the sentential forms generated by $G$.

* Supported by a CAPES PhD grant.
1. Obtain a completely specified deterministic finite automaton corresponding to the regular expression $R$. Note that the input alphabet for this automaton is $\{\text{lhs/rhs} : \text{lhs} \rightarrow \text{rhs} \in \mathcal{P}\}$.
2. Invert the orientation of all its transitions and swap the roles of its initial and final states. Convert the resulting machine into a deterministic finite-state automaton;
3. For each transition $p \in \delta(q, \text{lhs/rhs})$ in the automaton, add a new one $(p, 0, \text{lhs})$ to the set defining $\delta(q, \lambda, (\text{rhs})^R)$ in the stack automaton;
4. For all final states of the deterministic finite automaton (item 2 above), convert them into final states of the stack automaton;
5. Add a new a state to the stack automaton set of states and create a $\lambda$ transition from this state to the initial state of the deterministic finite automaton.

The resulting machine is a context-sensitive parser. Next we show an illustrative example.

**Example. Parser for the Triple-Balanced Language, Parsing the Sentence $a^3 b^3 c^3$.** The parser in this example has been constructed according to the strategy described above, based on the following context-free grammar with regular control set and without appearance checking $G = (\{S, A, B\}, \{a, b, c\}, \{p_1, p_2, p_3, p_4, p_5\}, S, p_1(p_2p_3)^*p_4p_5, \emptyset)$ where: $p_1 = S \rightarrow AB, p_2 = A \rightarrow aAb, p_3 = B \rightarrow cB, p_4 = A \rightarrow ab, p_5 = B \rightarrow c$.

![Fig. 1. A Stack Automaton that accepts $a^n b^n c^n$](image)
The table on the right traces the parsing process after pushing the input string onto the stack. The header marks the parsing time, counted from the moment the automaton reaches state $q_2$ on.

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>A</td>
<td>A</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>A</td>
<td>S</td>
</tr>
<tr>
<td>previous state</td>
<td>$q_0$</td>
<td>$q_2$</td>
<td>$q_4$</td>
<td>$q_5$</td>
<td>$q_4$</td>
<td>$q_5$</td>
<td>$q_4$</td>
<td>$q_4$</td>
</tr>
<tr>
<td>applied production</td>
<td>$p_5$</td>
<td>$p_3$</td>
<td>$p_5$</td>
<td>$p_3$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$p_2$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>current state</td>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_4$</td>
<td>$q_5$</td>
<td>$q_4$</td>
<td>$q_5$</td>
<td>$q_4$</td>
<td>$q_4$</td>
</tr>
</tbody>
</table>

References


Finite-State Molecular Computing

Olgierd Unold¹, Maciej Troć¹, Tadeusz Dobosz², and Alicja Trusewicz²

¹ Institute of Engineering Cybernetics, Wroclaw University of Technology
Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland
unold@ict.pwr.wroc.pl
http://www.ict.pwr.wroc.pl/~unold

² Institute of Forensic Medicine, Medical University
Mikulicza-Radeckiego 4, 50-368 Wroclaw, Poland
tdobosz@forensic.am.wroc.pl

Abstract. In this paper we explore the molecular computation model based on a splicing system and implemented in vitro by Shapiro. This paper presents two implementations, one implementing an FSA with two states, and the other one with three states, using enzymes BseMII and BseXI, respectively, which are different from only the one – FokI – used by Shapiro, et al. for their two-state FSA. The model of three-state, two-input symbol machine supports 1,835,001 syntactically distinct programs in comparison with 765 programs in Shapiro's approach.

Summary

Many DNA computing models have been proposed so far. Some of them have been actually implemented in in vitro experiments, but most remain mere proposals and their feasibility has not yet been verified. Regardless of applied DNA model in order to construct a general molecular computer some universal model of computation must be expressed in chemistry, such as a Turing Machine or its special cases including finite state automata (FSM). In this paper, we explore the computation model proposed by Ehud Shapiro et al [1], which is in fact a kind of splicing system [2]. Splicing is a paradigm for DNA Computing which provides a theoretical model of enzymatic systems operating on DNA strands. Splicing models a solution with enzymatic actions, i.e. restriction, hybridization and ligation, operating on DNA in parallel in a common test tube.

We have developed a simulator for a programmable two-state, two-input symbol finite automaton proposed by Shapiro and his associates [3]. The simulator is able to check all automata programs, not only used to test in a tube by Shapiro’s team. We modified molecular computing machine by using new restriction endonuclease – BseMII: CTCAG(N)₁₀/₈↓ where N ∈ {A,C,G,T}, and BseXI: GCAGC(N)(8/12)↓ where N ∈ {A,C,G,T}, and where A- adenine, G-guanine, C-cytosine, and T-thymine. The new encoding of an automaton was proposed.

The model of two-state automata was extended to the model of three-state, two-input symbol machine. This kind of the automaton can have 18 possible transition

© Springer-Verlag Berlin Heidelberg 2003
rules. There are 262,143 possible transition-rule selections and 7 possible selections of accepting states, resulting in 1,835,001 syntactically distinct programs. In our approach the enzyme BseMII was used. We have implemented and virtually checked automaton shown in Figure 1. The first set of experiments in vitro was started.

![Diagram of the 3-state example automaton. Incoming unlabelled arrow represents the initial state (S2), labeled arrows represent transition rules, and the double circle represents an accepting state (S0).]

1) sequence of the input symbols \(ba\) represented by the DNA strand
   CTCAG8CTACCATCCGCC300
   GAGTC8GATGTTAGGCGCG300
   Initial state: S2  Final state: S0
   2) restriction using BseMII:
   CTCAG8CT ACCATCCGCC300
   GAGTC8 GATGTTAGGCGCG300
   3) ligation with transition rule S2 \(\rightarrow\) S1 represented by the DNA strand
   CTCAG(3)CT
   GAGTC(3)
   4) restriction using BseMII:
   CTCAG3CTACCATCCGCC300
   GAGTC3GATGTTAGGCGCG300
   5) ligation with transition rule S1 \(\rightarrow\) a S0 represented by the DNA strand
   CTCAG(3)AT
   GAGTC(3)
   6) restriction using BseMII:
   CTCAG3ATCCGCC300
   GAGTC3TAGGCGCG300
   7) ligation with output detector S0-D represented by the DNA strand
   (100)GC
   (100)
   b) A sample output of the simulator for the input word \(ba\)

Fig. 1. Three-state, two-input molecular finite automaton

References

<table>
<thead>
<tr>
<th>Author Name</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allauzen, Cyril</td>
<td>83</td>
</tr>
<tr>
<td>Baeijs, Christof</td>
<td>108</td>
</tr>
<tr>
<td>Bravo, César</td>
<td>306</td>
</tr>
<tr>
<td>Bultan, Tevifik</td>
<td>188</td>
</tr>
<tr>
<td>Carson-Berndsen, Julie</td>
<td>301</td>
</tr>
<tr>
<td>Castaño, José M.</td>
<td>269</td>
</tr>
<tr>
<td>Champarnaud, Jean-Marc</td>
<td>49</td>
</tr>
<tr>
<td>Cleophas, Loek G.</td>
<td>304</td>
</tr>
<tr>
<td>Daley, Mark</td>
<td>239</td>
</tr>
<tr>
<td>Dang, Zhe</td>
<td>163</td>
</tr>
<tr>
<td>Darriba, Victor M.</td>
<td>258</td>
</tr>
<tr>
<td>Dobosz, Tadeusz</td>
<td>309</td>
</tr>
<tr>
<td>Eramian, Mark</td>
<td>239</td>
</tr>
<tr>
<td>Fan, Tsai-Hung</td>
<td>251</td>
</tr>
<tr>
<td>Frishert, Michiel</td>
<td>304</td>
</tr>
<tr>
<td>Fritz, Carsten</td>
<td>35</td>
</tr>
<tr>
<td>Fu, Xiang</td>
<td>188</td>
</tr>
<tr>
<td>Gaál, Tamás</td>
<td>108</td>
</tr>
<tr>
<td>Guingne, Franck</td>
<td>108, 131</td>
</tr>
<tr>
<td>Hagiya, Masami</td>
<td>213</td>
</tr>
<tr>
<td>Henzinger, Thomas A.</td>
<td>1</td>
</tr>
<tr>
<td>Hosoya, Haruo</td>
<td>201</td>
</tr>
<tr>
<td>Hütten, Patrick</td>
<td>11</td>
</tr>
<tr>
<td>Hwang, Geng-Dian</td>
<td>176</td>
</tr>
<tr>
<td>Inenaga, Shunsuke</td>
<td>120</td>
</tr>
<tr>
<td>Karhumäki, Juhani</td>
<td>3</td>
</tr>
<tr>
<td>Kelly, Robert</td>
<td>301</td>
</tr>
<tr>
<td>Kempe, André</td>
<td>108, 131</td>
</tr>
<tr>
<td>Kourie, Derrick G.</td>
<td>299</td>
</tr>
<tr>
<td>Krishnamoorthy, Mukkai S.</td>
<td>297</td>
</tr>
<tr>
<td>Kuske, Dietrich</td>
<td>150</td>
</tr>
<tr>
<td>Laugerotte, Éric</td>
<td>49</td>
</tr>
<tr>
<td>Lee, Shufen</td>
<td>251</td>
</tr>
<tr>
<td>Li, Cheng</td>
<td>163</td>
</tr>
<tr>
<td>Lombardy, Sylvain</td>
<td>96</td>
</tr>
<tr>
<td>Lu, Hsueh-I</td>
<td>251</td>
</tr>
<tr>
<td>McQuillan, Ian</td>
<td>239</td>
</tr>
<tr>
<td>Meinecke, Ingmar</td>
<td>150</td>
</tr>
<tr>
<td>Melichar, Bořivoj</td>
<td>294</td>
</tr>
<tr>
<td>Miyamoto, Satoru</td>
<td>120</td>
</tr>
<tr>
<td>Mohri, Mehryar</td>
<td>83</td>
</tr>
<tr>
<td>Murata, Makoto</td>
<td>201</td>
</tr>
<tr>
<td>Neto, João José</td>
<td>306</td>
</tr>
<tr>
<td>Ngassam, E. Ketcha</td>
<td>299</td>
</tr>
<tr>
<td>Nicart, Florent</td>
<td>108, 131</td>
</tr>
<tr>
<td>Ouardi, Faissal</td>
<td>49</td>
</tr>
<tr>
<td>Polcar, Tomáš</td>
<td>294</td>
</tr>
<tr>
<td>Poss, Raphaël</td>
<td>96</td>
</tr>
<tr>
<td>Preston, Nathan</td>
<td>297</td>
</tr>
<tr>
<td>Ravikumar, Bala</td>
<td>226</td>
</tr>
<tr>
<td>Régis-Gianas, Yann</td>
<td>96</td>
</tr>
<tr>
<td>Rodríguez, Leandro</td>
<td>258</td>
</tr>
<tr>
<td>Sakarovitch, Jacques</td>
<td>96</td>
</tr>
<tr>
<td>Shinohara, Ayumi</td>
<td>120</td>
</tr>
<tr>
<td>Su, Jianwen</td>
<td>188</td>
</tr>
<tr>
<td>Sutner, Klaus</td>
<td>282</td>
</tr>
<tr>
<td>Takeda, Masayuki</td>
<td>120</td>
</tr>
<tr>
<td>Tamn, Hellis</td>
<td>61</td>
</tr>
<tr>
<td>Thomas, Wolfgang</td>
<td>11</td>
</tr>
<tr>
<td>Tozawa, Akihiko</td>
<td>213</td>
</tr>
<tr>
<td>Trakhtman, Avraham</td>
<td>141</td>
</tr>
<tr>
<td>Troc, Maciej</td>
<td>309</td>
</tr>
<tr>
<td>Trusewicz, Alicja</td>
<td>309</td>
</tr>
<tr>
<td>Tsou, Tsung-Shan</td>
<td>251</td>
</tr>
<tr>
<td>Ukkonen, Esko</td>
<td>61</td>
</tr>
<tr>
<td>Unold, Olgierd</td>
<td>309</td>
</tr>
<tr>
<td>Vilares, Jesús</td>
<td>258</td>
</tr>
<tr>
<td>Vilares, Manuel</td>
<td>258</td>
</tr>
<tr>
<td>Wallmeier, Nico</td>
<td>11</td>
</tr>
<tr>
<td>Wang, Farn</td>
<td>23, 176</td>
</tr>
<tr>
<td>Wang, Tsai-Cheng</td>
<td>251</td>
</tr>
<tr>
<td>Watson, Bruce W.</td>
<td>299, 304</td>
</tr>
<tr>
<td>Xie, Gaoyan</td>
<td>163</td>
</tr>
<tr>
<td>Author</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>Yao, Adam</td>
<td>251</td>
</tr>
<tr>
<td>Yen, Hsu-Chun</td>
<td>23</td>
</tr>
<tr>
<td>Yu, Fang</td>
<td>176</td>
</tr>
<tr>
<td>Ziadi, Djelloul</td>
<td>49</td>
</tr>
<tr>
<td>Zijl, Lynette van</td>
<td>72</td>
</tr>
</tbody>
</table>