Lecture Notes in Computer Science 2088
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Sheng Yu Andrei Păun (Eds.)

Implementation and Application of Automata

5th International Conference, CIAA 2000
London, Ontario, Canada, July 24-25, 2000
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Sheng Yu
Andrei Păun
The University of Western Ontario, Department of Computer Science
Middlesex College, London, ON, Canada, N6A 5B7
E-mail: {syu,apaun}@csd.uwo.ca

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Implementation and application of automata : 5th international conference ; revised papers / CIAA 2000, London, Ontario, Canada, July 24 - 25, 2000 / Sheng Yu ; Andrei Paun (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2088)
ISBN 3-540-42491-1

CR Subject Classification (1998): F.1.1, F.4.3, F.3, F.2

ISSN 0302-9743
ISBN 3-540-42491-1 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10839354 06/3142 5 4 3 2 1 0

Foreword

The Fifth International Conference on Implementation and Application of Automata (CIAA 2000) was held at the University of Western Ontario in London, Ontario, Canada on July 24-25, 2000. This conference series was formerly called the International Workshop on Implementing Automata (WIA)

This volume of the Lecture Notes in Computer Science series contains all the papers that were presented at CIAA 2000, and also the abstracts of the poster papers that were displayed during the conference.

The conference addressed issues in automata application and implementation. The topics of the papers presented at this conference ranged from automata applications in software engineering, natural language and speech recognition, and image processing, to new representations and algorithms for efficient implementation of automata and related structures.

Automata theory is one of the oldest areas in computer science. Research in automata theory has always been motivated by its applications since its early stages of development. In the 1960s and 1970s, automata research was motivated heavily by problems arising from compiler construction, circuit design, string matching, etc. In recent years, many new applications have been found in various areas of computer science as well as in other disciplines. Examples of the new applications include statecharts in object-oriented modeling, finite transducers in natural language processing, and nondeterministic finite-state models in communication protocols. Many of the new applications do not and cannot simply apply the existing models and algorithms in automata theory to their problems. New models, or modifications of the existing models, are needed to satisfy their requirements. A feature that can be found in many of the new applications is that the sizes of the problems in practice are astronomically larger than those occuring in the traditional applications. New algorithms and new representations of automata are in demand to reduce the time and space requirements of the computation.

The CIAA conference series provides a forum for those new problems and new challenges. In these conferences, both theoretical and practical results related to application and implementation of automata can be presented and discussed, and software packages and toolkits can be demonstrated. The participants of the conference series have come from both research institutions and industry.

I wish to thank all the program committee members and referees for their efforts in refereeing and selecting papers. This volume is edited with much help from Andrei Păun and Mihaela Păun. Their efforts are very much appreciated.

We wish to thank EATCS and ACM SIGACT for their sponsorship and HyperVision for their donation to the conference. We also thank the editors of

the Lecture Notes in Computer Science series and Springer-Verlag, in particular Ms. Anna Kramer, for their help in publishing this volume.

May 2001 Sheng Yu

CIAA 2000 Program Committee:

Program Committee Chair:
Sheng Yu, University of Western Ontario, Canada

Jean-Marc Champarnaud	Université de Rouen, France
Maxime Crochemore	Université de Marne-la-Vallée, France
Franze Günthner	Ludwig Maximilian Universität München, Germany
Oscar Ibarra	University of California at Santa Barbara, USA
Helmut Jürgensen	University of Western Ontario, Canada
Lauri Karttunen	Xerox Palo Alto Research Center, USA
Denis Maurel	Université de Tours, France
Mehryar Mohri	AT&T Laboratories, USA
Jean-Eric Pin	Université de Paris VII, France
Wolfgang Thomas	Technical University of Aachen, Germany
Bruce Watson	University of Pretoria, South Africa
Derek Wood	Hong Kong University of Science and Technology, China

Tri-event Organizing Committee

John Hart	Helmut Jürgensen (Co-chair)
Lila Kari	Kai Salomaa
Stephen Watt	Sheng Yu (Co-chair)

Conference Coordinators

Victoria Stasiuk	Mihaela Păun

Volunteers

Rachel Bevan	Mark Daley
Mark G. Eramian	Sandy Huerter
Xiang Ji	Rob Kitto
Andrei Păun	Clement Renard
Mark Sumner	Geoff Wozniak

List of Referees

M.-P. Béal
J. Berstel
T. Bultan
J.-M. Champarnaud
M. Crochemore
K. Culik II
M. Daley
M.G. Eramian
F. Günthner
M. Holzer
S. Huerter
O. H. Ibarra
T. Jiang
H. Jürgensen
L. Kari
L. Karttunen
R. Kitto
R. Laporte
C. Löding
P. Madhusudan
D. Maurel
S. Michelin
M. Mohri
A. Păun
J.-E. Pin
G. Roussel
K. Salomaa
J. Senellart
W. Thomas
J. Vöge
B. Watson
D. Wood
S. Yu

Table of Contents

Invited Lectures

Technical Contributions

Abstracts

Synthesizing State-Based Object Systems from LSC Specifications

David Harel and Hillel Kugler

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
{harel,kugler}@wisdom.weizmann.ac.il

Abstract. Live sequence charts (LSCs) have been defined recently as an extension of message sequence charts (MSCs; or their UML variant, sequence diagrams) for rich inter-object specification. One of the main additions is the notion of universal charts and hot, mandatory behavior, which, among other things, enables one to specify forbidden scenarios. LSCs are thus essentially as expressive as statecharts. This paper deals with synthesis, which is the problem of deciding, given an LSC specification, if there exists a satisfying object system and, if so, to synthesize one automatically. The synthesis problem is crucial in the development of complex systems, since sequence diagrams serve as the manifestation of use cases — whether used formally or informally — and if synthesizable they could lead directly to implementation. Synthesis is considerably harder for LSCs than for MSCs, and we tackle it by defining consistency, showing that an entire LSC specification is consistent iff it is satisfiable by a state-based object system, and then synthesizing a satisfying system as a collection of finite state machines or statecharts.

1 Introduction

1.1 Background and Motivation

Message sequence charts (MSCs) are a popular means for specifying scenarios that capture the communication between processes or objects. They are particularly useful in the early stages of system development. MSCs have found their way into many methodologies, and are also a part of the UML [UMLdocs], where they are called **sequence diagrams**. There is also a standard for the MSC language, which has appeared as a recommendation of the ITU [Z120] (previously called the CCITT).

Damm and Harel [DH99] have raised a few problematic issues regarding MSCs, most notably some severe limitations in their expressive power. The semantics of the language is a rather weak partial ordering of events. It can be used to make sure that the sending and receiving of messages, if occurring, happens in the right order, but very little can be said about what the system actually does, how it behaves when false conditions are encountered, and which scenarios are forbidden. This weakness prevents sequence charts from becoming a serious

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 1–33, 2001.

means for describing system behavior, e.g., as an adequate language for substantiating the use-cases of [J92,UMLdocs]. Damm and Harel [DH99] then go on to define **live sequence charts** (**LSC**s), as a rather rich extension of MSCs. The main addition is **liveness**, or **universality**, which provides constructs for specifying not only possible behavior, but also necessary, or mandatory behavior, both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time within a chart. Liveness allows for the specification of "anti-scenarios" (forbidden ones), and strengthens structuring constructs like as subcharts, branching and iteration. LSCs are essentially as expressive as statecharts. As explained in [DH99], the new language can serve as the basis of tools supporting specification and analysis of use-cases and scenarios — both formally and informally — thus providing a far more powerful means for setting requirements for complex systems.

The availability of a scenario-oriented language with this kind of expressive power is also a prerequisite to addressing one of the central problems in behavioral specification of systems: (in the words of [DH99]) to relate scenario-based inter-object specification with state machine intra-object specification. One of the most pressing issues in relating these two dual approaches to specifying behavior is **synthesis** , i.e., the problem of automatically constructing a behaviorally equivalent state-based specification from the scenarios. Specifically, we want to be able to generate a statechart for each object from an LSC specification of the system, if this is possible in principle. The synthesis problem is crucial in the development of complex object-oriented systems, since sequence diagrams serve to instantiate use cases. If we can synthesize state-based systems from them, we can use tools such as Rhapsody (see [HG97]) to generate running code directly from them, and we will have taken a most significant step towards going automatically from instantiated use-cases to implementation, which is an exciting (and ambitious!) possibility. See the discussion in the recent [H00]. And, of course, we couldn't have said this about the (far easier) problem of synthesizing from conventional sequence diagrams, or MSCs, since their limited expressive power would render the synthesized system too weak to be really useful; in particular, there would be no way to guarantee that the synthesized system would satisfy safety constraints (i.e., that bad things — such as a missile firing with the radar not locked on the target — will not happen).

In this paper we address the synthesis problem in a slightly restricted LSC language, and for an object model in which behavior of objects is described by state machines with synchronous communication. For the most part the resulting state machines are orthogonality-free and flat, but in the last section of the paper we sketch a construction that takes advantage of the more advanced constructs of statecharts.

An important point to be made is that the most interesting and difficult aspects in the development of complex systems stem from the interaction between different features, which in our case is modeled by the requirements made in different charts. Hence, a synthesis approach that deals only with a single chart — even if it is an LSC — does not solve the crux of the problem.

The paper is organized as follows. Section 2 introduces the railcar system of [HG97] and shows how it can be specified using LSCs. This example will be used throughout the paper to explain and illustrate our main ideas. Section 3 then goes on to explain the LSC semantics and to define when an object system satisfies an LSC specification. In Section 4 we define the **consistency** of an LSC specification and prove that consistency is a necessary and sufficient condition for satisfiability. We then describe an algorithm for deciding if a given specification is consistent. The synthesis problem is addressed in Section 5, where we present a synthesis algorithm that assumes fairness. We then go on to show how this algorithm can be extended to systems that do not guarantee fairness. (Lacking fairness, the system synthesized does not generate the most general language as it does in the presence of fairness.) In Section 6 we outline an algorithm for synthesizing statecharts, with their concurrent, orthogonal state components.

1.2 Related Work

As far as the limited case of classical message sequence charts goes, there has been quite some work on synthesis from them. This includes the SCED method [KM94,KSTM98] and synthesis in the framework of ROOM charts [LMR98]. Other relevant work appears in [SD93,AY99,AEY00,BK98,KGSB99,WS00]. The full paper [HKg99] provides brief descriptions of these efforts.[1] In addition, there is the work described in [KW00], which deals with LSCs, but synthesizes from a single chart only: an LSC is translated into a timed Büchi automaton (from which code can be derived).

In addition to synthesis work directly from sequence diagrams of one kind or another, one should realize that constructing a program from a specification is a long-known general and fundamental problem. There has been much research on constructing a program from a specification given in temporal logic.

The early work on this kind of synthesis considered *closed systems*, that do not interact with the environment [MW80,EC82]. In this case a program can be extracted from a constructive proof that the formula is satisfiable. This approach is not suited to synthesizing *open systems* that interact with the environment, since satisfiability implies the existence of an environment in which the program satisfies the formula, but the synthesized program cannot restrict the environment. Later work in [PR89a,PR89b,ALW89,WD91] dealt with the synthesis of open systems from linear temporal logic specifications. The realizability problem is reduced to checking the nonemptiness of tree automata, and a finite state program can be synthesized from an infinite tree accepted by the automaton.

In [PR90], synthesis of a distributed reactive system is considered. Given an architecture — a set of processors and their interconnection scheme — a solution to the synthesis problem yields finite state programs, one for each processor, whose joint behavior satisfies the specification. It is shown in [PR90] that the realizability of a given specification over a given architecture is undecidable.

[1] See technical report MCS99-20, October 1999, The Weizmann Institute of Science, at http://www.wisdom.weizmann.ac.il/reports.html.

Previous work assumed the easy architecture of a single processor, and then realizability was decidable. In our work, an object of the synthesized system can share all the information it has with all other objects, so the undecidability results of [PR90] do not apply here.

Another important approach discussed in [PR90] is first synthesizing a single processor program, and then decomposing it to yield a set of programs for the different processors. The problem of finite-state decomposition is an easier problem than realizing an implementation. Indeed, it is shown in [PR90] that decompositionality of a given finite state program into a set of programs over a given architecture is decidable. The construction we present in Section 5 can be viewed as following parts of this approach by initially synthesizing a global system automaton describing the behavior of the entire system and then distributing it, yielding a set of state machines, one for each object in the system. However, the work on temporal logic synthesis assumes a model in which the system and the environment take turns making moves, each side making one move in its turn. We consider a more realistic model, in which after each move by the environment, the system can make any finite number of moves before the environment makes its next move.

2 An Example

In this section we introduce the railcar system, which will be used throughout the paper as an example to explain and illustrate the main ideas and results. A detailed description of the system appears in [HG97], while [DH99] uses it to illustrate LSC specifications. To make this paper self contained and to illustrate the main ideas of LSCs, we now show some of the basic objects and scenarios of the example.

The automated railcar system consists of six terminals, located on a cyclic path. Each pair of adjacent terminals is connected by two rail tracks. Several railcars are available to transport passengers between terminals.

Here now is some of the required behavior, using LSC's. Fig. 1 describes a car departing from a terminal. The objects participating in this scenario are **cruiser**, **car**, **carHandler**. The chart describes the message communication between the objects, with time propagating from top to bottom. The chart of Fig. 1 is universal. Whenever its activation message occurs, i.e., the car receives the message **setDest** from the environment, the sequence of messages in the chart should occur in the following order: the car sends a departure request **departReq** to the car handler, which sends a departure acknowledgment **departAck** back to the car. The car then sends a **start** message to the cruiser in order to activate the engine, and the cruiser responds by sending **started** to the car. Finally, the car sends **engage** to the cruiser and now the car can depart from the terminal.

A scenario in which a car approaches the terminal is described in Fig. 2. This chart is also universal, but here instead of having a single message as an activation, the chart is activated by the prechart shown in the upper part of the figure (in dashed line-style, and looking like a condition, since it is conditional

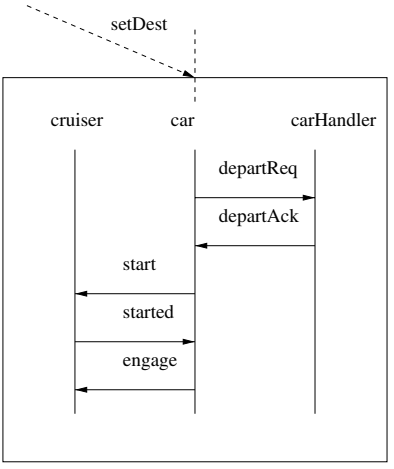


Fig. 1. Perform Departure

in the cold sense of the word — a notion we explain below): in the prechart, the message **departAck** is communicated between the car handler and the car, and the message **alert100** is communicated between the proximity sensor and the car. If these messages indeed occur as specified in the prechart, then the body of the chart must hold: the car sends the arrival request **arrivReq** to the car handler, which sends an arrival acknowledgment **arrivAck** back to the car.

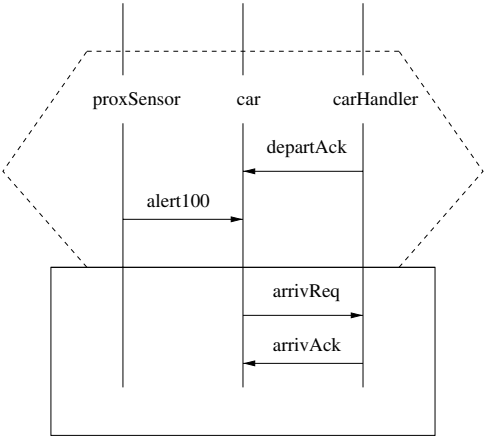


Fig. 2. Perform Approach

Figs. 3 and 4 are existential charts, depicted by dashed borderlines. These charts describe two possible scenarios of a car approaching a terminal: stop at

terminal and pass through terminal, respectively. Since the charts are existential, they need not be satisfied in all runs; it is only required that for each of these charts the system has at least one run satisfying it. In an iterative development of LSC specifications, such existential charts may be considered informal, or underspecified, and can later be transformed into universal charts specifying the exact activation message or prechart that is to determine when each of the possible approaches happens.

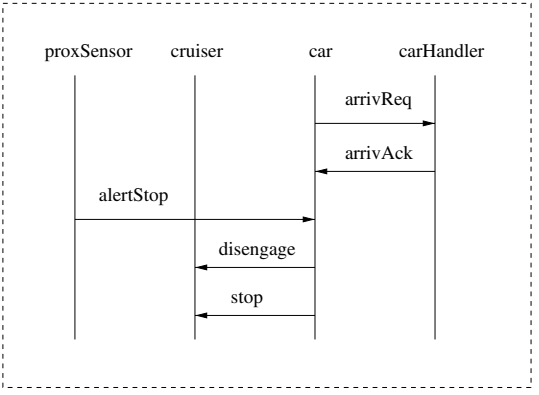


Fig. 3. Stop at terminal

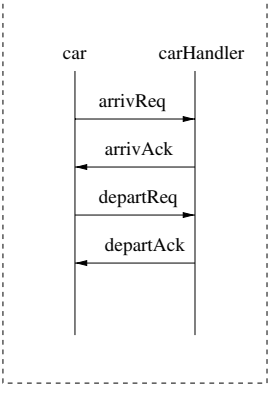


Fig. 4. Pass through terminal

The simple universal chart in Fig. 5 requires that when the proximity sensor receives the message **comingClose** from the environment, signifying that the car is getting close to the terminal, it sends the message **alert100** to the car.

This prevents a system from satisfying the chart in Fig. 2 by never sending the message **alert100** from the proximity sensor to the car, so that the prechart is never satisfied and there is no requirement that the body of the chart hold.

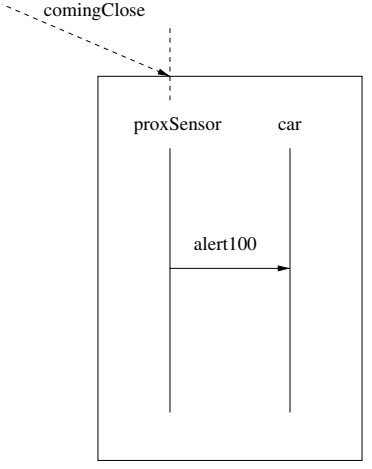


Fig. 5. Coming close to terminal

The set of charts in Figs. 1–5 can be considered as an LSC specification for (part of) the railcar system. Our goal in this paper is to develop algorithms to decide, for any such specification, if there is a satisfying object system and, if so, to synthesize one automatically. As mentioned in the introduction, what makes our goal both harder and more interesting is in the treatment of a set of charts, not just a single one.

3 LSC Semantics

The semantics of the LSC language is defined in [DH99], and we now explain some of the basic definitions and concepts of this semantics using the railcar example.

Consider the **Perform Departure** chart of Fig. 1. In Fig. 6 it appears with a labeling of the **locations** of the chart. The set of locations for this chart is thus:

$$\{\langle cruiser, 0\rangle, \langle cruiser, 1\rangle, \langle cruiser, 2\rangle, \langle cruiser, 3\rangle, \langle car, 0\rangle, \langle car, 1\rangle, \langle car, 2\rangle,$$
$$\langle car, 3\rangle, \langle car, 4\rangle, \langle car, 5\rangle, \langle carHandler, 0\rangle, \langle carHandler, 1\rangle, \langle carHandler, 2\rangle\}$$

The chart defines a partial order $<_m$ on locations. The requirement for order along an instance line implies, for example, $\langle car, 0\rangle <_m \langle car, 1\rangle$. The order induced from message sending implies, for example, $\langle car, 1\rangle <_m \langle carHandler, 1\rangle$. From transitivity we get that $\langle car, 0\rangle <_m \langle carHandler, 1\rangle$.

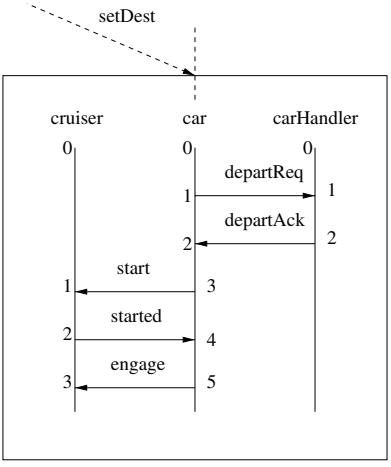


Fig. 6.

One of the basic concepts used for defining the semantics of LSCs, and later on in our synthesis algorithms, is the notion of a **cut**. A cut through a chart represents the progress each instance has made in the scenario. Not every "slice", i.e., a set consisting of one location for each instance, is a cut. For example,

$$(\langle cruiser, 1\rangle, \langle car, 2\rangle, \langle carHandler, 2\rangle)$$

is not a cut. Intuitively the reason for this is that to receive the message **start** by the cruiser (in location $\langle cruiser, 1\rangle$), the message must have been sent, so location $\langle car, 3\rangle$ must have already been reached.

The cuts for the chart of Fig. 7 are thus:

$\{(\langle cruiser, 0\rangle, \langle car, 0\rangle, \langle carHandler, 0\rangle), (\langle cruiser, 0\rangle, \langle car, 1\rangle, \langle carHandler, 0\rangle),$
$(\langle cruiser, 0\rangle, \langle car, 1\rangle, \langle carHandler, 1\rangle), (\langle cruiser, 0\rangle, \langle car, 1\rangle, \langle carHandler, 2\rangle),$
$(\langle cruiser, 0\rangle, \langle car, 2\rangle, \langle carHandler, 2\rangle), (\langle cruiser, 0\rangle, \langle car, 3\rangle, \langle carHandler, 2\rangle),$
$(\langle cruiser, 1\rangle, \langle car, 3\rangle, \langle carHandler, 2\rangle), (\langle cruiser, 2\rangle, \langle car, 3\rangle, \langle carHandler, 2\rangle),$
$(\langle cruiser, 2\rangle, \langle car, 4\rangle, \langle carHandler, 2\rangle), (\langle cruiser, 2\rangle, \langle car, 5\rangle, \langle carHandler, 2\rangle),$
$(\langle cruiser, 3\rangle, \langle car, 5\rangle, \langle carHandler, 2\rangle)\}$

The sequence of cuts in this order constitutes a **run**. The **trace** of this run is:

$(env, car.setDest), (car, carHandler.departReq), (carHandler, car.departAck),$
$(car, cruiser.start), (cruiser, car.started), (car, cruiser.engage)$

This chart has only one run, but in general a chart can have many runs. Consider the chart in Fig. 7. From the initial cut $(0, 0, 0, 0)$ [2] it is possible to

[2] We often omit the names of the objects, for simplicity, when listing cuts.

progress either by the car sending **departReq** to the car handler, or by the passenger sending **pressButton** to the destPanel. Similarly there are possible choices from other cuts. Fig. 8 gives an automaton representation for all the possible runs. This will be the basic idea for the construction of the synthesized state machines in our synthesis algorithms later on. Each state, except for the special starting state s_0, represents a cut and is labeled by the vector of locations. Successor cuts are connected by edges labeled with the message sent. Assuming a synchronous model we do not have separate edges for the sending and receiving of the same message. A path starting from s_0 that returns to s_0 represents a run.

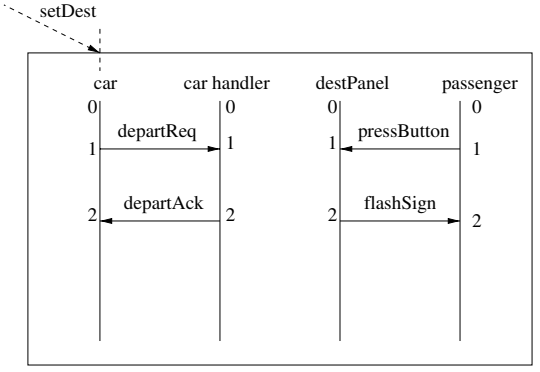


Fig. 7.

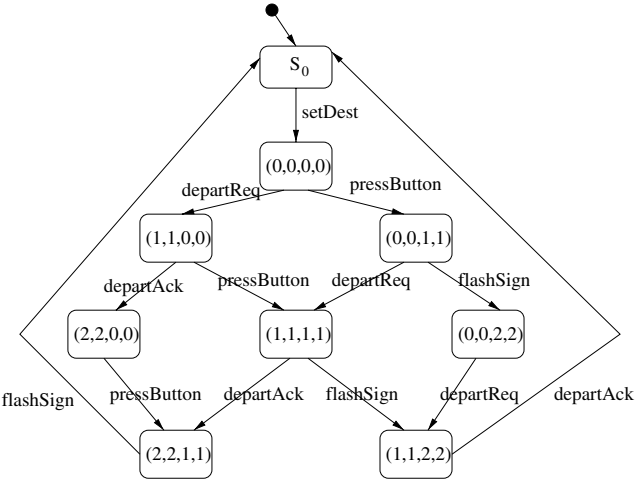


Fig. 8.

Here are two sample traces from these runs:

$(env, car.setDest), (car, carHandler.departReq), (carHandler, car.departAck),$
$(passenger, destPanel.pressButton), (destPanel, passenger.flashSign)$

(env, car.setDest), (car, carHandler.departReq), (passenger, destPanel. pressButton),
(carHandler, car.departAck), (destPanel, passenger.flashSign)

As part of the "liveness" extensions, the LSC language enables forcing progress along an instance line. Each location is given a temperature **hot** or **cold**, graphically denoted by solid or dashed segments of the instance line. A run *must* continue down solid lines, while it *may* continue down dashed lines. Formally, we require that in the final cut in a run all locations are **cold**. Consider the **perform approach** scenario appearing in Fig. 9. The dashed segments in the lower part of the car and carHandler instances specify that it is possible that the message **arrivAck** will not be sent, even in a run in which the prechart holds. This might happen in a situation where the terminal is closed or when all the platforms are full.

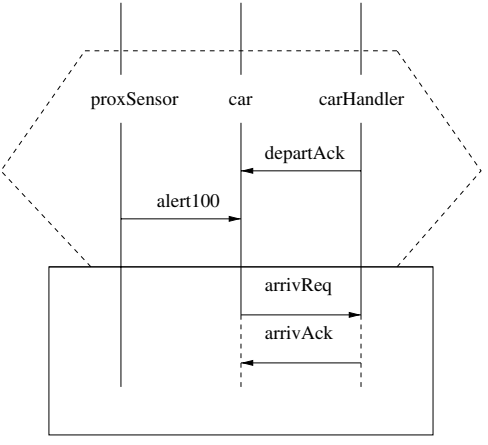


Fig. 9.

When defining the languages of a chart in [DH99], messages that do not appear in the chart are not restricted and are allowed to occur in-between the messages that do appear, without violating the chart. This is an abstraction mechanism that enables concentrating on the relevant messages in a scenario. In practice it may be useful to restrict messages that do not appear explicitly in the chart. Each chart will then have a designated set of messages that are not allowed to occur anywhere except if specified explicitly in the chart; and this applies even

if they do not appear anywhere in the chart. A tool may support convenient selection of this message set. Consider the **perform departure** scenario in Fig. 1. By taking its set of messages to include those appearing therein, but also **alert100**, **arrivReq** and **arrivAck**, we restrict these three messages from occurring during the departure scenario, which makes sense since we cannot arrive to a terminal when we are just in the middle of departing from one.

As in [DH99], an **LSC specification** is defined as:

$$LS = \langle M, amsg, mod \rangle,$$

where M is a set of charts, and $amsg$ and mod are the activation messages[3] and the modes of the charts (existential or universal), respectively.

A system **satisfies** an LSC specification if, for every universal chart and every run, whenever the activation message holds the run must satisfy the chart, and if, for every existential chart, there is at least one run in which the activation message holds and then the chart is satisfied. Formally,

Definition 1. *A system S* **satisfies** *the LSC specification $LS = \langle M, amsg, mod \rangle$, written $S \models LS$, if:*

1. $\forall m \in M, \quad mod(m) = universal \Rightarrow \forall \eta \; \mathcal{L}_S^\eta \subseteq \mathcal{L}_m$
2. $\forall m \in M, \quad mod(m) = existential \Rightarrow \exists \eta \; \mathcal{L}_S^\eta \cap \mathcal{L}_m \neq \emptyset$

Here $\mathcal{L}_S^\eta$ is the trace set of object system S on the sequence of directed requests η. We omit a detailed definition here, which can be found, e.g., in [HKp99]. $\mathcal{L}_m$ is the language of the chart m, containing all traces satisfying the chart. We say that an LSC specification is **satisfiable** if there is a system that satisfies it.

4 Consistency of LSCs

Our goal is to automatically construct an object system that is correct with respect to a given LSC specification. When working with an expressive language like LSCs that enables specifying both necessary and forbidden behavior, and in which a specification is a well-defined set of charts of different kinds, there might very well be self contradictions, so that there might be no object system that satisfies it.

Consider an LSC specification that contains the universal charts of Figs. 10 and 11. The message **setDest** sent from the environment to the car activates Fig. 10, which requires that following the **departReq** message, **departAck** is sent from the car handler to the car. This message activates Fig. 11, which requires the sending of **engage** from the car to the cruiser before the **start** and **started** messages are sent, while Fig. 10 requires the opposite ordering. A contradiction.

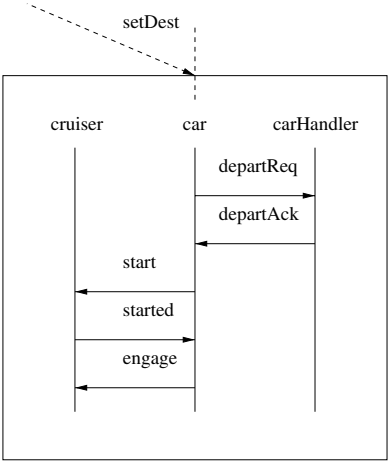


Fig. 10.

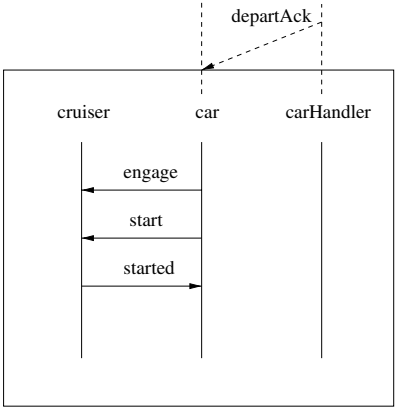


Fig. 11.

This is only a simple example of an inconsistency in an LSC specification. Inconsistencies can be caused by such an "interaction" between more than two universal charts, and also when a scenario described in an existential chart can never occur because of the restrictions from the universal charts. In a complicated system consisting of many charts the task of finding such inconsistencies manually by the developers can be formidable, and algorithmic support for this process can help in overcoming major problems in early stages of the analysis.

[3] In the general case we allow a prechart instead of only a single activation message. However, in this paper we provide the proofs of our results for activation messages, but they can be generalized rather easily to precharts too.

4.1 Consistency = Satisfiability

We now provide a global notion of the consistency of an LSC specification. This is easy to do for conventional, existential MSCs, but is harder for LSCs. In particular, we have to make sure that a universal chart is satisfied by *all* runs, from *all* points in time.

We will use the following notation: A_{in} is the alphabet denoting messages sent from the environment to objects in the system, while A_{out} denotes messages sent between the objects in the system.

Definition 2. *An LSC specification* $LS = \langle M, amsg, mod \rangle$ *is* **consistent** *if there exists a nonempty regular language* $\mathcal{L}_1 \subseteq (A_{in} \cdot A_{out}^*)^*$ *satisfying the following properties:*

1. $\mathcal{L}_1 \subseteq \bigcap_{m_j \in M,\ mod(m_j)=universal} \mathcal{L}_{m_j}$
2. $\forall w \in \mathcal{L}_1\ \forall a \in A_{in}\ \exists r \in A_{out}^*,\ \ s.t.\ \ w \cdot a \cdot r \in \mathcal{L}_1.$
3. $\forall w \in \mathcal{L}_1,\ w = x \cdot y \cdot z,\ y \in A_{in} \Rightarrow x \in \mathcal{L}_1.$
4. $\forall m \in M,\ mod(m) = existential \Rightarrow \mathcal{L}_m \cap \mathcal{L}_1 \neq \emptyset.$

The language $\mathcal{L}_1$ is what we require as the set of satisfying traces. Clause 1 in the definition requires all universal charts to be satisfied by all the traces in $\mathcal{L}_1$, Clause 2 requires a trace to be extendible if a new message is sent in from the environment, Clause 3 essentially requires traces to be completed before new messages from the environment are dealt with, and Clause 4 requires existential charts to be satisfied by traces from within $\mathcal{L}_1$.

Now comes the first central result of the paper, showing that the consistency of an LSC specification is a necessary and sufficient condition for the existence of an object system satisfying it.

Theorem 1. *A specification LS is satisfiable if and only if it is consistent.*

Proof. Appears in the Appendix.

A basic concept used in the proof of Theorem 1 is the notion of a **global system automaton**, or a **GSA**. A GSA describes the behavior of the entire system — the message communication between the objects in the system in response to messages received from the environment. A rigorous definition of the GSA appears in the appendix. Basically, it is a finite state automaton with input alphabet consisting of messages sent from the environment to the system (A_{in}), and output alphabet consisting of messages communicated between the objects in the system (A_{out}). The GSA may have **null transitions**, transitions that can be taken spontaneously without the triggering of a message. We add a **fairness requirement**: a null transition that is enabled an infinite number of times must be taken an infinite number of times. A *fair cycle* is a loop of states connected by null transitions, which can be taken repeatedly without violating

the fairness requirement. We require that the system has no fair cycles, thus ensuring that the system's reactions are finite.

In the proof of Theorem 1 (the *Consistency* $\Rightarrow$ *Satisfiability* direction in the Appendix) we show that it is possible to construct a GSA satisfying the specification. This implies the existence of an object system (a separate automaton for each object) satisfying the specification. Later on, when discussing synthesis, we will show methods for the distribution of the GSA between the objects to obtain a satisfying object system. In section 5.5 we show that the fairness requirement is not essential for our construction — it is possible to synthesize a satisfying object system that does not use null transition and the fairness requirement, although it does not generate the most general language.

4.2 Deciding Consistency

It follows from Theorem 1 that to prove the existence of an object system satisfying an LSC specification LS, it suffices to prove that LS is consistent. In this section we present an algorithm for deciding consistency.

A basic construction used in the algorithm is that of a deterministic finite automaton accepting the language of a universal chart. Such an automaton for the chart of Fig. 7 is shown in Fig. 12. The initial state s_0 is the only accepting state. The activation message **setDest** causes a transition from state s_0, and the automaton will return to s_0 only if the messages **departReq, departAck, pressButton** and **flashSign** occur as specified in the chart. Notice that the different orderings of these messages that are allowed by the chart are represented in the automaton by different paths. Each such message causes a transition to a state representing a successor cut. The self transitions of the nonaccepting states allow only messages that are not restricted by the chart. The initial state s_0 has self transitions for message **comingClose** sent from the environment and for all other messages between objects in the system. To avoid cluttering the figure we have not written the messages on the self transitions.

The construction algorithm of this automaton and its proof of correctness are omitted from this version of the paper.

An automaton accepting exactly the runs that satisfy *all* the universal charts can be constructed by intersecting these separate automata. This intersection automaton will be used in the algorithm for deciding consistency. The idea is to start with this automaton, which represents the "largest" regular language satisfying all the universal charts, and to systematically narrow it down in order to avoid states from which the system will be forced by the environment to violate the specification. At the end we must check that there are still representative runs satisfying each of the existential charts.

Here, now is our algorithm for checking consistency:

Algorithm 2 *1. Find the minimal DFA* $\mathcal{A} = (A, S, s_0, \rho, F)$ *that accepts the language*

$$\mathcal{L} = \bigcap_{m_j \in M,\ mod(m_j)=universal} \mathcal{L}_{m_j}$$

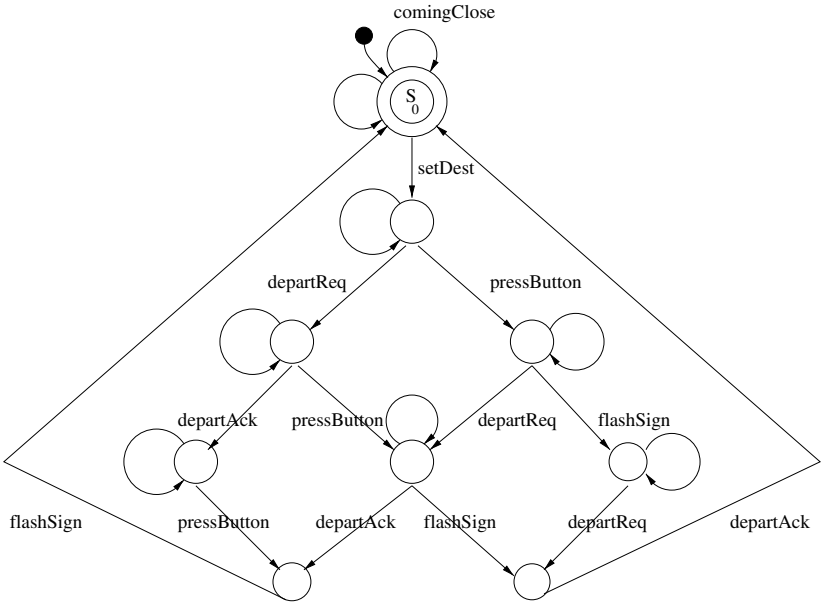


Fig. 12.

(The existence of such an automaton follows from the discussion above and is proved in the full version of the paper [HKg99].)

2. *Define the sets* $Bad_i \subseteq S$, *for* $i = 0, 1, ...,$ *as follows:*
   $Bad_0 = \{s \in S \mid \exists a \in A_{in},\ s.t.\ \forall x \in A^*_{out}\ \rho(s, a \cdot x) \notin F\}$,
   $Bad_i = \{s \in S \mid \exists a \in A_{in},\ s.t.\ \forall x \in A^*_{out}\ \rho(s, a \cdot x) \notin F - Bad_{i-1}\}$.
   The series Bad_i *is monotonically increasing, with* $Bad_i \subseteq Bad_{i+1}$, *and since* S *is finite it converges. Let us denote the limit set by* Bad_{max}.
3. *From* $\mathcal{A}$ *define a new automaton* $\mathcal{A}' = (A, S, s_0, \rho, F')$, *where the set of accepting states has been reduced to* $F' = F - Bad_{max}$
4. *Further reduce* $\mathcal{A}$, *by removing all transitions that lead from states in* $S - F'$, *and which are labeled with elements of* A_{in}. *This yields the new automaton* $\mathcal{A}''$.
5. *Check whether* $\mathcal{L}(\mathcal{A}'') \neq \emptyset$ *and whether, in addition,* $\mathcal{L}_{m_i} \cap \mathcal{L}(\mathcal{A}'') \neq \emptyset$ *for each* $m_i \in M$ *with* $mod(m_i) = existential$. *If both are true output* YES*; otherwise output* NO.

Proposition 1. *The algorithm is correct: given a specification LS, it terminates and outputs* YES *iff LS is consistent.*

Proof. Omitted in this version of the paper. See [HKg99].

In case the algorithm answers YES, the specification is consistent and it is possible to proceed to automatically synthesize the system, as we show in the

next section. However, for the cases where the algorithm answers NO, it would be very helpful to provide the developer with information about the source of the inconsistency. Step 5 of our algorithm provides the basis for achieving this goal. Here is how.

The answer is NO if $\mathcal{L}(\mathcal{A}'') = \emptyset$ or if there is an existential chart m_i such that $\mathcal{L}_{m_i} \cap \mathcal{L}(\mathcal{A}'') = \emptyset$. In the second case, this existential chart is the information we need. The first case is more delicate: there is a sequence of messages sent from the environment to the system (possibly depending on the reactions of the system) that eventually causes the system to violate the specification. Unlike verification against a specification, where we are given a specific program or system and can display a specific run of it as a counter-example, here we want to **synthesize** the object system so we do not yet have any concrete runs. A possible solution is to let the supporting tool play the environment and the user play the system, with the aim of locating the inconsistency. The tool can display the charts graphically and highlight the messages sent and the progress made in the different charts. After each message sent by the environment (determined by the tool using the information obtained in the consistency algorithm), the user decides which messages are sent between the objects in the system. The tool can suggest a possible reaction of the system, and allow the user to modify it or choose a different one. Eventually, a universal chart will be violated, and the chart and the exact location of this violation can be displayed.

5 Synthesis of FSMs from LSCs

We now show how to automatically synthesize a satisfying object system from a given consistent specification. We first use the algorithm for deciding consistency (Algorithm 2), relying on the equivalence of consistency and satisfiability (Theorem 1) to derive a global system automaton, a GSA, satisfying the specification. Synthesis then proceeds by distributing this automaton between the objects, creating a desired object system.

The synthesis is demonstrated on our example, taking the charts in Figs. 1–5 to be the required LSC specification. For the universal charts, Figs. 1, 2 and 5, we assume that the sets of restricted messages (those not appearing in the charts) are { **alertStop**, **alert100**, **arrivReq**, **arrivAck**, **disengage**, **stop** }, { **departReq**, **start**, **started**, **engage** } and { **departAck**}, respectively.

Figs. 13, 14 and 15 show the automata for the **perform departure**, **perform approach** and **coming close** charts, respectively. Notice that in Fig. 14 there are two accepting states s_0 and s_1, since we have a prechart with messages **departAck** and **alert100** that causes activation of the body of the chart. To avoid cluttering the figures we have not written the messages on the self transitions. For nonaccepting states these messages are the non-restricted messages between objects in the system, while for accepting states we take all messages that do not cause a transition from the state, including messages sent by the environment.

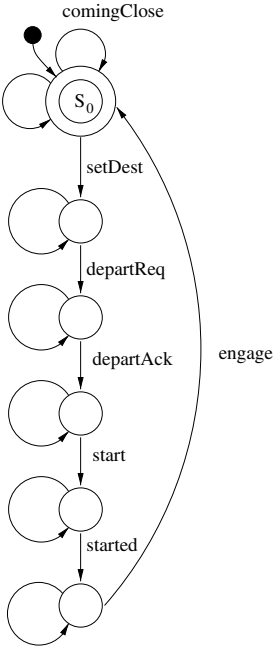


Fig. 13. Automaton for Perform Departure

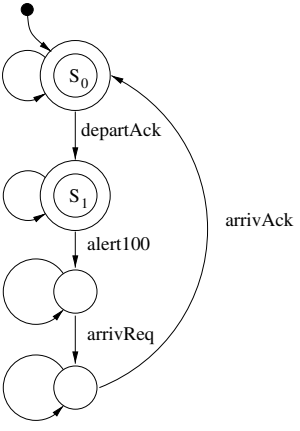


Fig. 14. Automaton for Perform Approach

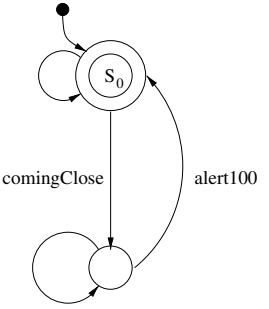


Fig. 15. Automaton for Coming Close

The intersection of the three automata of Figs. 13, 14 and 15 is shown in Fig. 16. It accepts all the runs that satisfy all three universal charts of our system.

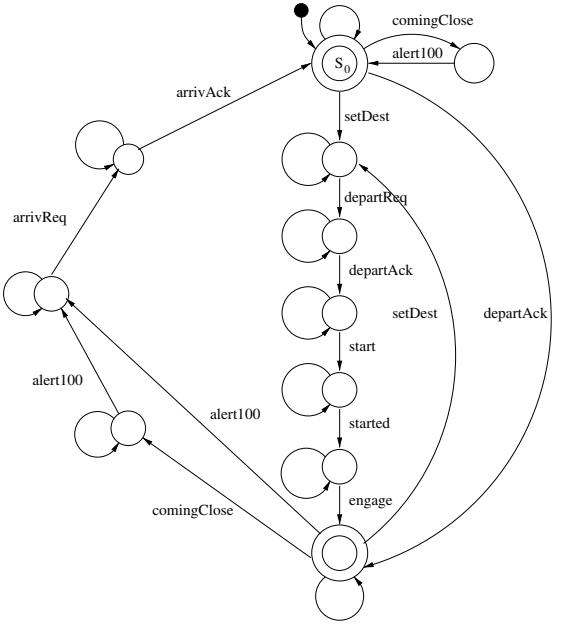


Fig. 16. The Intersection Automaton

The global system automaton (GSA) derived from this intersection automaton is shown in Fig. 17. The two accepting states have as outgoing transitions only messages from the environment. This has been achieved using the techniques described in the proof of Theorem 1 (see the Appendix). Notice also the

existence of runs satisfying each of the existential charts. We have used the path extraction methods of Section 5.5 to retain these runs.

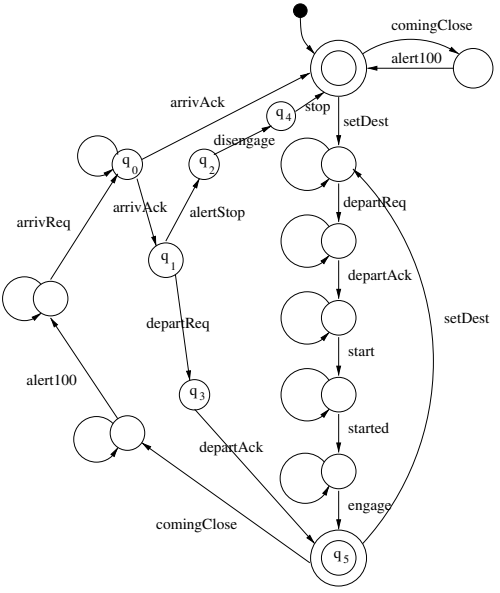


Fig. 17. The Global System Automaton

After constructing the GSA, the synthesis proceeds by distributing the automaton between the objects, creating a desired object system. To illustrate the distribution we focus on a subautomaton of the GSA consisting of the states q_0, q_1, q_2, q_3, q_4 and q_5, as appearing in Fig. 17. This subautomaton is shown in Fig. 18. In this figure we provide full information about the message, the sender and receiver, since this information is important for the distribution process.

In general, let $A = \langle Q, q_0, \delta \rangle$ be a GSA describing a system with objects $\mathcal{O} = \{O_1, ..., O_n\}$ and messages $\Sigma = \Sigma_{in} \cup \Sigma_{out}$. Assume that A satisfies the LSC specification LS. Our constructions employ new messages taken from a set Σ_{col}, where $\Sigma_{col} \cap \Sigma = \emptyset$. They will be used by the objects for collaboration in order to satisfy the specification, and are not restricted by the charts in LS.

There are different ways to distribute the global system automaton between the objects. In the next three subsections we discuss three main approaches — **controller object**, **full duplication**, and **partial duplication** — and illustrate them on the GSA subautomaton of Fig. 18. The first approach is trivial and is shown essentially just to complete the proof of the existence of an object system satisfying a consistent specification. The second method is an intermediate stage towards the third approach, which is more realistic.

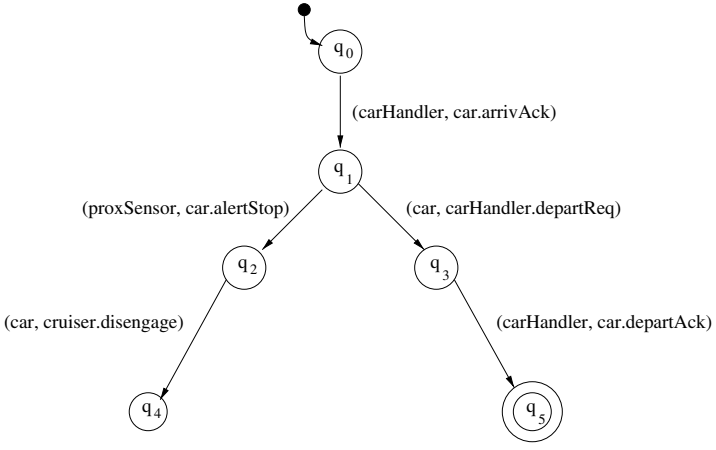


Fig. 18. Subautomaton of the GSA

5.1 Controller Object

In this approach we add to the set of objects in the system $\mathcal{O}$ an additional object O_{con} which acts as the controller of the system, sending commands to all the other objects. These will have simple automata to enable them to carry out the commands.

Let $|\Sigma_{col}| = |A_{in}| + |A_{out}|$, and let f be a one-to-one function

$$f : A_{in} \cup A_{out} \to \Sigma_{col}$$

We define the state machine of the controller object O_{con} to be $\langle Q, q_0, \delta_{con} \rangle$, and the state machines of object $O_i \in \mathcal{O}$ to be $\langle \{q_{O_i}\}, q_{O_i}, \delta_{O_i} \rangle$.

The states and the initial state of O_{con} are identical to those of the GSA. The transition relation δ_{con} and the transition relations δ_{O_i} are defined as follows:

If $(q, a, q') \in \delta$ where $a \in A_{in}$, $a = (env, O_i.\sigma_i)$ then

$$(q, f(a), q') \in \delta_{con} \quad \text{and} \quad (q_{O_i}, \sigma_i / O_{con}.f(a), q_{O_i}) \in \delta_{O_i}.$$

If $(q, /a, q') \in \delta$ where $a \in A_{out}$, $a = (O_i, O_j.\sigma_j)$ then

$$(q, /O_i.f(a), q') \in \delta_{con} \quad \text{and} \quad (q_{O_i}, f(a)/O_j.\sigma_j, q_{O_i}) \in \delta_{O^i}.$$

This construction is illustrated in Fig. 19, which shows the object system obtained by the synthesis from the GSA of Fig. 18. It includes the state machine of the controller object O_{con}, and the transitions of the single-state state machines of the objects **carHandler**, **car** and **proxSensor**.

The size of the state machine of the controller object O_{con} is equal to that of the GSA, while all other objects have state machines with one state. Section 5.4 discusses the total complexity of the construction.

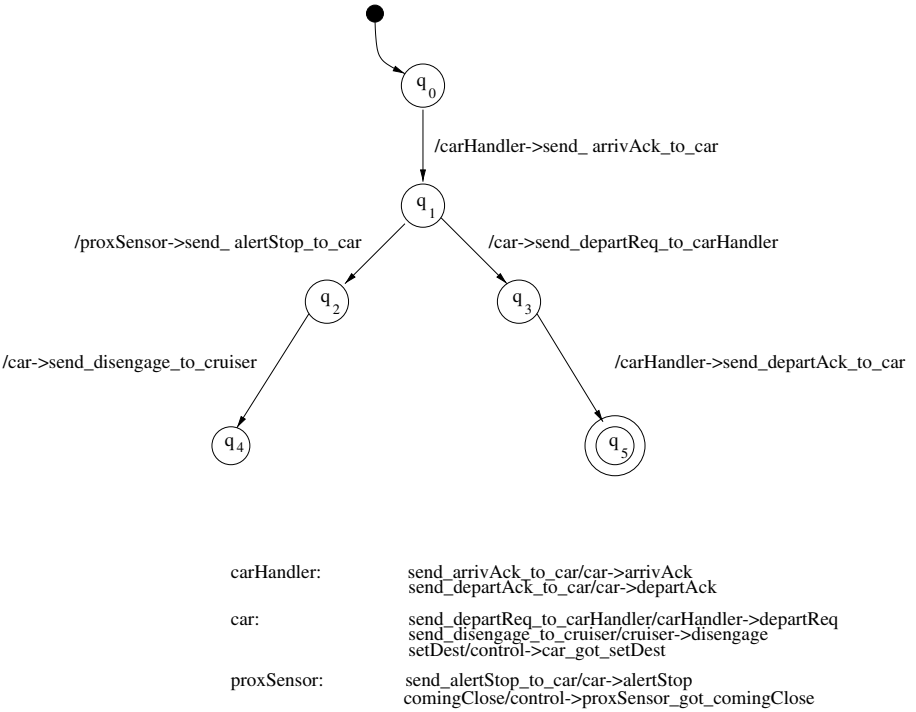


Fig. 19. Controller

5.2 Full Duplication

In this construction there is no controller object. Instead, each object will have the state structure of the GSA, and will thus "know" what state the GSA would have been in.

Recalling that $A = \langle Q, q_0, \delta \rangle$ is the GSA, let k be the maximum outdegree of the states in Q. A labeling of the transitions of A is a one-to-one function tn:

$$tn : \delta \to \{1, ..., k\}$$

Let $|\Sigma_{col}| = k$ and let f be a one-to-one function

$$f : \{1, ..., k\} \to \Sigma_{col}$$

The state machine for object O_i in $\mathcal{O}$ is defined to be $\langle Q, q_0, \delta_{O_i} \rangle$.
If $(q, a, q') \in \delta$, where $a \in A_{in}$, $a = (env, O_i.\sigma_i)$ and $a' = f(tn(q, a, q')) \in \Sigma_{col}$, then $(q, \sigma_i / O_{i+1}.a') \in \delta_{O_i}$ and for every $j \neq i$, $(q, a' / O_{j+1}.a', q') \in \delta_{O_j}$.
If $(q, /a, q') \in \delta$, where $a \in A_{out}$, $a = (O_i, O_j.\sigma_j)$ and $a' = f(tn(q, /a, q')) \in \Sigma_{col}$, then $(q, /O_j.\sigma_j; O_{i+1}.a', q') \in \delta_{O_i}$ and for every $j \neq i$, $(q, a' / O_{j+1}.a', q') \in \delta_{O_j}$.

This construction is illustrated in Fig. 20 on the sub-GSA of Fig. 18. The maximal outdegree of the states of the GSA in this example is 2, and the set of

collaboration messages is $\Sigma_{col} = \{1, 2\}$. Again, complexity is discussed in Section 5.4.

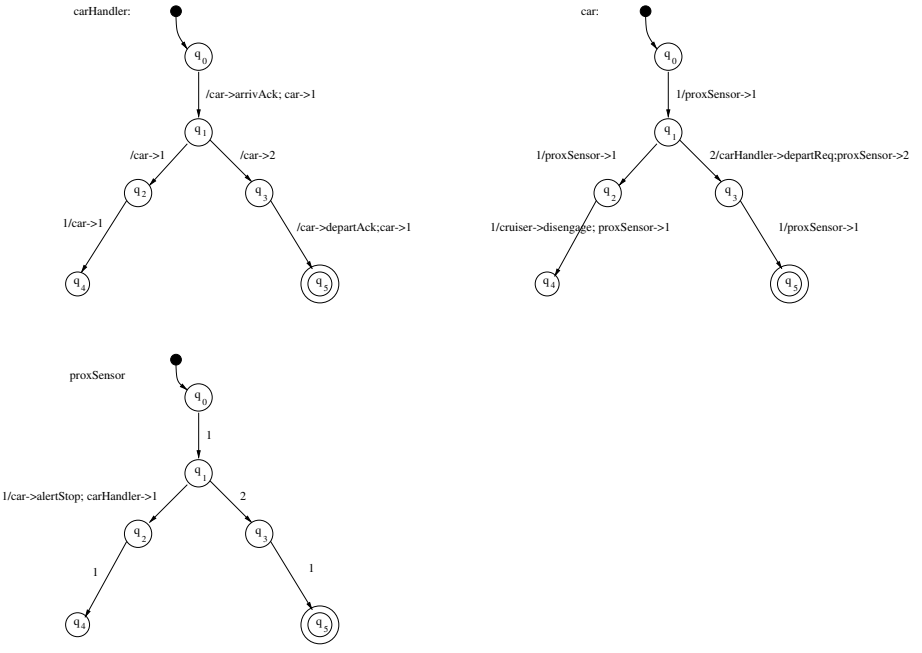


Fig. 20. Full Duplication

5.3 Partial Duplication

The idea here is to distribute the GSA as in the full duplication construction, but to merge states that carry information that is not relevant to this object in question. In some cases this can reduce the total size, although the worst case complexity remains the same.

The state machine of object O_i is defined to be $\langle Q_{O_i} \cup q_{idle}, q_0, \delta_{O_i} \rangle$, where $Q_{O_i} \subseteq Q$ is defined by

$$Q_{O_i} = \left\{ q \in Q \;\middle|\; \begin{array}{l} \exists q' \in Q \; \exists a \in A_{out} \text{ s.t.} \\ a = (O_i, O_j.\sigma_j), (q, /a, q') \in \delta \text{ or} \\ \exists q' \in Q \; \exists a \in A_{in} \text{ s.t.} \\ a = (env, O_i.\sigma_i), (q', a, q) \in \delta \end{array} \right\}$$

Thus, in object O_i we keep the states that the GSA enters after receiving a message from the environment, and the states from which O_i sends messages.

Let $|\Sigma_{col}| = |Q|$, and let f be a one-to-one function

$$f : Q \to \Sigma_{col}$$

The transition relation δ_{O_i} for object O_i is defined as follows:
If $(q, a, q') \in \delta$, $a = (env, O_i.\sigma_i)$ then $(q, \sigma_i / O_{i+1}.f(q'), q') \in \delta_{O_i}$.
If $(q, /a, q') \in \delta$, $a = (O_j, O_i.\sigma_i)$, then
either $q' \in Q_{O_j}$ and then $(q, /O_i.\sigma_i; O_{j+1}.f(q'), q') \in \delta_{O_j}$,
or $q' \notin Q_{O_j}$ and then $(q, /O_i.\sigma_i; O_{j+1}.f(q'), q_{idle}) \in \delta_{O_j}$.
If $q \in Q_{O_i}$, $q' \in Q_{O_i}$ then $(q, f(q'), q') \in \delta_{O_i}$.
If $q \in Q_{O_i}$, $q' \notin Q_{O_i}$ then $(q, f(q'), q_{idle}) \in \delta_{O_i}$.
For every $q \in Q_{O_i}$, $(q_{idle}, f(q), q) \in \delta_{O_i}$.

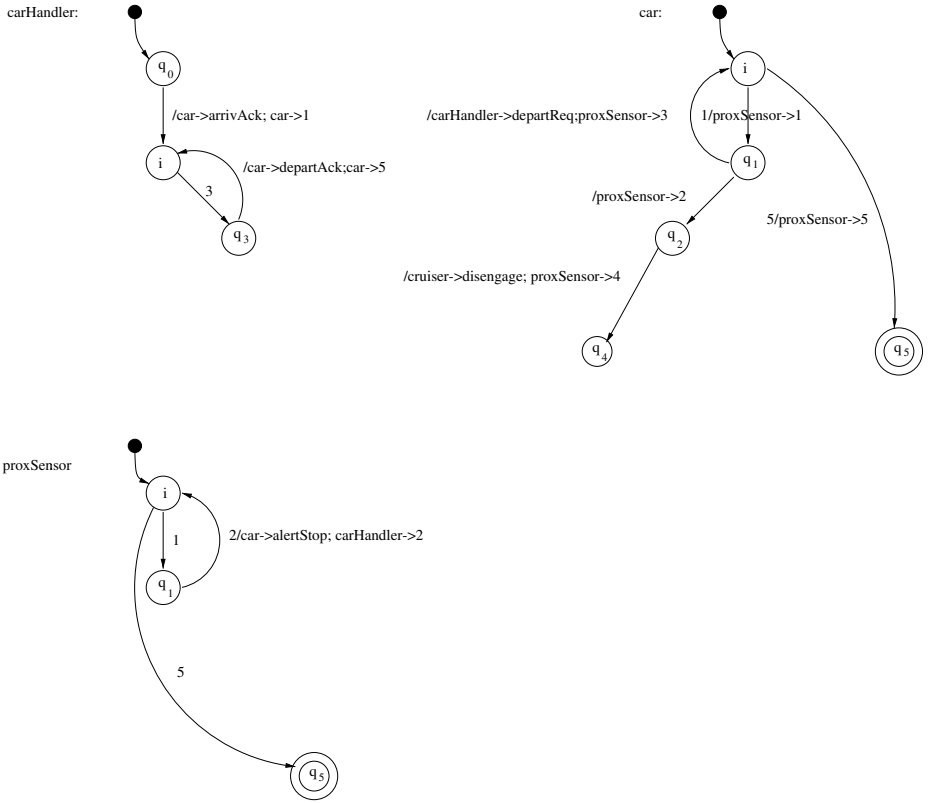


Fig. 21. Partial Duplication

This construction is illustrated in Fig. 21. The states of the GSA of Fig. 18 that were eliminated are q_1, q_2, q_4 and q_5 for **carHandler**, q_0 and q_3 for **car** and q_0, q_2, q_3 and q_4 for **proxSensor**.

5.4 Complexity Issues

In the previous sections we showed how to distribute the satisfying GSA between the objects, to create an object system satifying the LSC specification LS. We now discuss the size of the resulting system, relative to that of LS.

We take the size of an LSC chart m to be

$$|m| = |dom(m)| = |\#\text{ of locations in } m|,$$

and the size of an LSC specification $LS = \langle M, amsg, mod \rangle$ to be

$$|LS| = \sum_{m \in M} |m|.$$

The size of the GSA $A = \langle Q, q_0, \delta \rangle$ is simply the number of states $|Q|$. We ignore the size of the transition function δ which is polynomial in the number of states $|Q|$. Similarly, the size of an object is the number of states in its state machine.

Let LS be a consistent specification, where the universal charts in M are $\{m_1, m_2, ..., m_t\}$. Let A be the satisfying GSA derived using the algorithm for deciding consistency (Algorithm 2). A was obtained by intersecting the automata $A_1, A_2, ..., A_t$ that accept the runs of charts $m_1, m_2, ..., m_t$, respectively, and then performing additional transformations that do not change the number of states in A. The states of automaton A_i correspond to the cuts through chart m_i, as illustrated, for example, in Fig. 9.

Proposition 2. *The number of cuts through a chart m with n instances is bounded by $|m|^n$.*

Proof. Omitted in this version of the paper.

This is consistent with the estimate given in [AY99] for their analysis of the complexity of model checking for MSCs. We now estimate the size of the GSA.

Proposition 3. *The size of the GSA automaton A constructed in the proof of Theorem 1 satisfies*

$$|A| \quad \leq \quad \prod_{i=1}^{t} |m_i|^{n_i} \quad \leq \quad |LS|^{nt},$$

where n_i is the number of instances appearing in chart m_i, n is the total number of instances appearing in LS, and t is the number of universal charts in LS.

Proof. Omitted in this version of the paper.

The size of the GSA A is thus polynomial in the size of the specification LS, if we are willing to treat the number of objects in the system and the number of charts in the specification as constants. In some cases a more realistic assumption would be to fix one of these two, in which case the synthesized

automaton would be exponential in the remaining one. The time complexity of the synthesis algorithm is polynomial in the size of A.

The size of the synthesized object system is determined by the size of the GSA A. In the controller object approach (Section 5.1), the controller object is of size $|A|$ and each of the other objects is of constant size (one state). In the full duplication approach (Section 5.2), each of the objects is of size $|A|$, while in the partial duplication approach (Section 5.3), the size of each of the objects can be smaller than $|A|$, but the total size of the system is at least $|A|$.

5.5 Synthesis without Fairness Assumptions

We have shown that for a consistent specification we can find a GSA and then construct an object system that satisfies the specification. This construction used null transitions and a fairness assumption related to them, i.e., that a null transition that is enabled an infinite number of times is taken an infinite number of times. We now show that consistent specifications also have satisfying object systems with no null transitions.

Let $A = \langle Q, q_0, \delta \rangle$ be the GSA satisfying the specification LS, derived using the algorithm for deciding consistency. We partition Q into two sets: Q_{stable}, the states in Q that do not have outgoing null transitions, and $Q_{transient}$, the states of which all the outgoing transitions are null transitions. Such a partition is possible, as implied by the proof of Theorem 1. Now, A may have a loop of null transitions consisting of states in $Q_{transient}$. Such a loop represents an infinite number of paths and it will not be possible to maintain all of them in a GSA without null transitions. To overcome this, we create a new GSA $A' = \langle Q', q_0, \delta' \rangle$, with $Q' \subseteq Q$ and $\delta' \subseteq \delta$, as follows.

Let $m \in M$ be an existential chart, $mod(m) = existential$. A satisfies the specification LS, so there exists a word w, with $w \in \mathcal{L}_A \cap \mathcal{L}_m$. Let $q^0, q^1, ...$ be the sequence of states that A goes through when generating w, and let $\delta^0, \delta^1 ...$ be the transitions taken. Since $w \in \mathcal{L}_m$, let $i_1, ..., i_k$, such that $w_{i_1} \cdot w_{i_2} \cdots w_{i_k} \in \mathcal{L}_m^{trc}$. Let j be the minimal index such that $j > i_k$ and $q^j \in Q_{stable}$. The new GSA A' will retain all the states that appear in the sequence $q^0, ..., q^j$ and all the transitions that were used in $\delta^0, ..., \delta^j$. This is done for every existential chart $m \in M$.

In addition, for every $q_i, q_j \in Q$ and for every $a \in A_{in}$, if there exists a sequence of states $q_i, q^1, ..., q^l, q_j$ such that $(q_i, a, q^1) \in \delta$ and for every $1 \leq k < l$ there is a null transition $\delta^k \in \delta$ between q^k and q^{k+1}, then for one such sequence we keep in A' the states $q^1, ..., q^k$ and the transitions $\delta^1, ..., \delta^k$.

All other states and transition of A are eliminated in going from A to A'.

Proposition 4. *The GSA A' satisfies the specification LS.*

Proof. Omitted in this version of the paper.

6 Synthesizing Statecharts

We now outline an approach for a synthesis algorithm that uses the main succinctness feature of statecharts [H87] (see also [HG97]), namely, concurrency, via orthogonal states.

Consider a consistent specification $LS = \langle M, amsg, mod \rangle$, where the universal charts in M are $M_{universal} = \{m_1, m_2, ..., m_t\}$. In the synthesized object system each object O_i will have a top-level AND state with t orthogonal component OR states, $s_1, s_2, ..., s_t$. Each s_j has substates corresponding to the locations of object O_i in chart m_j.

Assume that in a scenario described by one of the charts in $M_{universal}$, object O_i has to send message σ to object O_j. If object O_i is in a state describing a location just before this sending, O_i will check whether O_j is in a state corresponding to the right location, and is ready to receive. (This can be done using one of the mechanisms of statecharts for sensing another object's state.) The message σ can then be sent and the transition taken. All the other component states of O_i and O_j will synchronously take the corresponding transitions if necessary.

This was a description of the local check that an object has to perform before sending a message and advancing to the next location for one chart. Actually, the story is more complicated, since when advancing in one chart we must check that this does not contradict anything in any of the other charts. Even more significantly, we also must check that taking the transition will not get the system into a situation in which it will not be able to satisfy one of the universal charts.

To deal with these issues the synthesis algorithm will have to figure out which state configurations should be avoided. Specifically, let c_i be a cut through chart m_i. We say that $C = (c_1, c_2, ..., c_t)$ is a **supercut** if for every i, c_i is a cut through m_i. We say that supercut $C' = (c'_1, c'_2, ..., c'_t)$ is a **successor** of supercut $C = (c_1, c_2, ..., c_t)$, if there exists i with $succ_{m_i}(c_i, (j, l_j), c'_i)$ and such that for all $k \neq i$ the cut c'_k is consistent with communicating the message $msg(j, l_j)$ while in cut c_k.
Now, for $i = 0, 1, ...$, define the sets

$Bad_i \subseteq \{$all supercuts s.t. at least one of the cuts has at least one hot location$\}$

as follows:

$$Bad_0 = \{C \mid C \text{ has no successors } \}$$

$$Bad_i = \{C | C \in Bad_{i-1} \text{ or all successors of } C \text{ are in } Bad_{i-1}\}$$

The series Bad_i is monotonically increasing under set inclusion, so that $Bad_i \subseteq Bad_{i+1}$. Since the set of all supercuts is finite the series converges. Denote its limit by Bad_{max}. The point now is that before taking a transition the synthesized object system will have to check that it does not lead to a supercut in Bad_{max}.

The construction is illustrated in Figs. 22, 23, 24 and 25 which show the statecharts for car, carHandler, proxSensor and cruiser, respectively, obtained

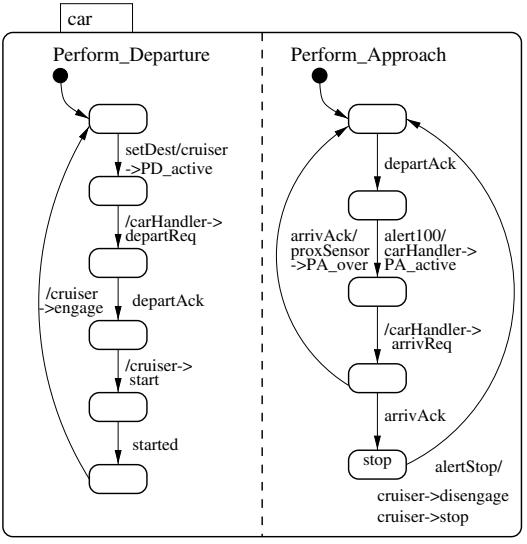


Fig. 22. Statechart of car

from the railcar system specification. Notice that an object that does not actively participate in some universal chart, does not need a component in its statechart for this scenario, for example proxSensor does not have a **Perform Departure** component. Notice the use of the *in* predicate in the statechart of the proxSensor for sensing if the car is in the **stop** state.

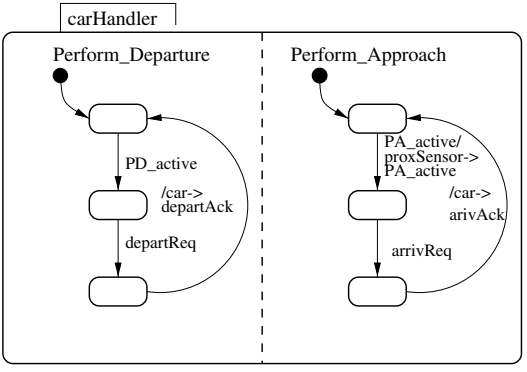


Fig. 23. Statechart of carHandler

The number of states in this kind of synthesized statechart-based system is on the order of the total number of locations in the charts of the specification. Now, although in the GSA solution the number of states was exponential in the number

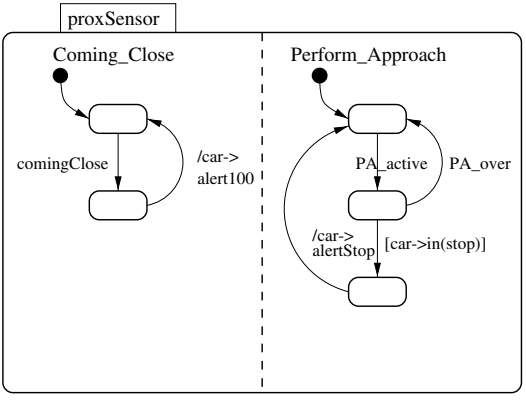


Fig. 24. Statechart of proxSensor

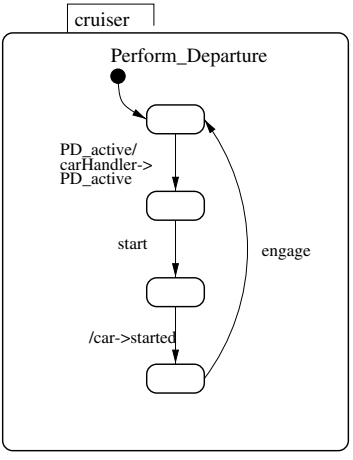


Fig. 25. Statechart of cruiser

of universal charts and in the number of objects in the system, which seems to contrast sharply with the situation here, the comparison is misleading; the guards of the transitions here may involve lengthy conditions on the states of the system. In practice, it may prove useful to use OBDD's for efficient representation and manipulation of conditions over the system state space.

Acknowledgements. We would like to thank Orna Kupferman, Tarja Systa and Erkki Mäkinen for their insightful comments on an early version of the paper.

References

[ABS91] Amon, T., G. Boriello and C. Sequin, "Operation/Event Graphs: A design representation for timing behavior", *Proc. '91 Conf. on Computer Hardware Description Language*, pp. 241 – 260, 1991.

[ALW89] Abadi, M., L. Lamport and P. Wolper, "Realizable and unrealizable concurrent program specifications", *Proc. 16th Int. Colloq. on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, pp. 1–17, 1989.

[AY99] Alur, R., and M. Yannakakis, "Model Checking of Message Sequence Charts", to appear in *Proc. 10th Int. Conf. on Concurrency Theory (CONCUR'99)*, Eindhoven, Netherlands, August 1999.

[AEY00] Alur, R., K. Etessami and M. Yannakakis, "Inference of Message Sequence Charts", *Proc. 22nd Int. Conf. on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.

[B88] Boriello, G., "A new interface specification methodology and its application to transducer synthesis", Technical Report UCB/CSD 88/430, University of California Berkeley, 1988.

[BK76] Biermann, A.W., and R. Krishnaswamy, "Constructing Programs from Example Computations", *IEEE Trans. Softw. Eng.*, SE-2 (1976), 141–153.

[BK98] Broy, M., and I. Krüger, "Interaction Interfaces — Towards a Scientific Foundation of a Methodological Usage of Message Sequence Charts", In *Formal Engineering Methods*, (J. Staples, M. G. Hinchey, and Shaoying Liu, eds), IEEE Computer Society, 1998, pp. 2–15.

[DH99] Damm, W., and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems*, (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, pp. 293–312, 1999.

[E90] Emerson, E.A., "Temporal and modal logic", *Handbook of Theoretical Computer Science*, (1990), 997–1072.

[EC82] Emerson, E.A., and E.M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons", *Sci. Comp. Prog.* **2** (1982), 241–266.

[H87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version: Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)

[H00] Harel, D., "From Play-In Scenarios To Code: An Achievable Dream", *Proc. Fundamental Approaches to Software Engineering* (*FASE*), Lecture Notes in Computer Science, Vol. 1783 (Tom Maibaum, ed.), Springer-Verlag, March 2000, pp. 22-34.

[HG97] Harel, D., and E. Gery, "Executable Object Modeling with Statecharts", *IEEE Computer*, (1997), 31–42.

[HKg99] Harel, D., and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications", 1999, Available as Technical Report MCS99-20, The Weizmann Institute of Science, Department of Computer Science, Rehovot, Israel, http://www.wisdom.weizmann.ac.il/reports.html.

[HKp99] Harel, D., and O. Kupferman, "On the Inheritance of State-Based Object Behavior", *Proc. 34th Int. Conf. on Component and Object Technology*, IEEE Computer Society, Santa Barbara, CA, August 2000.

[J92] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA, 1992.

[KGSB99] Krüger, I., R. Grosu, P. Scholz and M. Broy "From MSCs to Statecharts", *Proc. DIPES'98*, Kluwer, 1999.

[KM94] Koskimies, K., and E. Makinen, "Automatic Synthesis of State Machines from Trace Diagrams", *Software — Practice and Experience* **24**:7 (1994), 643–658.

[KSTM98] Koskimies, K., T. Systa, J. Tuomi and T. Mannisto, "Automated Support for Modeling OO Software", *IEEE Software* **15**:1 (1998), 87–94.

[KV97] Kupferman, O., and M.Y. Vardi, "Synthesis with incomplete information", *Proc. 2nd Int. Conf. on Temporal Logic (ICTL97)*, pp. 91–106, 1997.

[KW00] Klose, J., and H. Wittke, "Automata Representation of Live Sequence Charts", manuscript, 2000.

[LMR98] Leue, S., L. Mehrmann and M. Rezai, "Synthesizing ROOM Models from Message Sequence Chart Specifications", University of Waterloo Tech. Report 98-06, April 1998.

[MW80] Manna, Z., and R.J. Waldinger, "A deductive approach to program synthesis" *ACM Trans. Prog. Lang. Sys.* **2** (1980), 90–121.

[PR89a] Pnueli, A., and R. Rosner, "On the Synthesis of a Reactive Module", *Proc. 16th ACM Symp. on Principles of Programming Languages*, Austin, TX, January 1989.

[PR89b] Pnueli, A., R. Rosner, "On the Synthesis of an Asynchronous Reactive Module", *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, pp. 652–671, 1989.

[PR90] Pnueli, A., R. Rosner, "Distributed Reactive Systems are Hard to Synthesize", *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pp. 746–757, 1990.

[SD93] Schlor, R. and W. Damm, "Specification and verification of system-level hardware designs using timing diagrams", *Proc. European Conference on Design Automation*, Paris, France, IEEE Computer Society Press, pp. 518 – 524, 1993.

[SGW94] Selic, B., G. Gullekson and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

[UMLdocs] Documentation of the Unified Modeling Language (UML), available from the Object Management Group (OMG), http://www.omg.org.

[WD91] Wong-Toi, H., and D.L. Dill, "Synthesizing processes and schedulers from temporal specifications", In *Computer-Aided Verification '90* (Clark and Kurshan, eds.), *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 3, pp. 177–186, 1991.

[WS00] Whittle, J. and J. Schumann, "Generating Statechart Designs from Scenarios", *Proc. 22nd Int. Conf. on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.

[Z120] Z.120 ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.

APPENDIX: Proof of Theorem 1

Proof. The proof relies on the definitions of an object system appearing in [HKp99], somewhat modified. In [HKp99] a basic computational model for object-oriented designs is presented. It defines the behavior of systems composed of instances of object classes, whose behavior is given by conventional state machines. In our work we assume a single instance of each class during the entire evolution of the system — we do not deal with dynamic creation and destruction of instances. We assume all messages are synchronous and that there are no failures in the system — every message that is sent is received.

($\Rightarrow$) Let the object system S be such that $S \models LS$. We let $\mathcal{L}_S = \cup_{\eta \in A_{in}^*} \mathcal{L}_S^\eta$, and show that $\mathcal{L}_S$ satisfies the four requirements of $\mathcal{L}_1$ in the definition of a consistent specification, Def. 2.

(1) From the definition of an object system it follows that $\mathcal{L}_S$ is regular and nonempty. The system S satisfies the specification LS. Hence, if we set $\mathcal{L} = \bigcap_{m_j \in M, mod(m_j)=universal} \mathcal{L}_{m_j}$, Clause 1 of the definition of satisfaction (Def. 1) implies $\forall \eta\ \mathcal{L}_S^\eta \subseteq \mathcal{L}$. Thus, $\mathcal{L}_S = \cup_\eta \mathcal{L}_S^\eta \subseteq \mathcal{L}$.

(2 and 3) Let $w \in \mathcal{L}_S$. There exists a sequence of directed requests sent by the environment, $\eta = O^0.\sigma^0 \cdot O^1.\sigma^1 \cdots O^n.\sigma^n$, such that w is the behavior of the system S while reacting to the sequence of requests η. Now, w belongs to the trace set of S on η, so that $w = w_0 \cdot w_1 \cdots w_n$, $w_i \in A^*$, $first(w_i) = (env, O^i.\sigma^i)$, and there exists a sequence of stable configurations $c_0, c_1, ..., c_{n+1}$ such that c_0 is initial and for all $0 \leq i \leq n$, $leads(c_i, w_i, c_{i+1})$. The *leads* predicate is defined in [HKp99]. It describes the reaction of the system to a message sent from the environment to the system that causes a transition of the system from the stable configuration c_i to a new stable configuration c_{i+1}, passing through a set of unstable configurations. The trace describing this behavior is w_i.

As to Clause 2 in Def. 2, the system reaches the stable configuration c_{n+1} at the end of the reactions to η. For any object O_i and request σ, there is a reaction of the system to the directed request $O_i.\sigma$ from the stable configuration c_{n+1}. If we denote by w_{n+1} the word that captures such a reaction, w_{n+1} is in the trace set of S on $O_i.\sigma$ from c_{n+1}, from which we obtain $w \cdot w_{n+1} \in \mathcal{L}_S$.

For Clause 3, assuming that $w = x \cdot y \cdot z$, $y \in A_{in}$, there exists i with $x = w_0 \cdots w_i$ and therefore $x \in \mathcal{L}_S$.

(4) The system S satisfies the specification LS. Hence, from Clause 2 of Def. 1 we have

$$\forall m \in M, mod(m) = existential \Rightarrow \exists \eta\ \mathcal{L}_S^\eta \cap \mathcal{L}_m \neq \emptyset$$

Since $\mathcal{L}_S^\eta \subseteq \mathcal{L}_S = \cup_\eta \mathcal{L}_S^\eta$, we obtain

$$\forall m \in M, mod(m) = existential \Rightarrow \mathcal{L}_S \cap \mathcal{L}_m \neq \emptyset$$

($\Leftarrow$) Let LS be consistent. We have to show that there exists an object system S satisfying LS. To prove this we define the notion of a **global system automaton**, or a **GSA**. We will show that there exists a GSA satisfying the specification and that it can be used to construct an object system satisfying LS.

A GSA A describing a system with objects $\mathcal{O} = \{O_1, ..., O_n\}$ and message set
$\Sigma = \Sigma_{in} \cup \Sigma_{out}$ is a tuple $A = \langle Q, q_0, \delta \rangle$, where Q is a finite set of states, q_0 is the initial state, and $\delta \subseteq Q \times \mathcal{B} \times Q$ is a transition relation. Here $\mathcal{B}$ is a set of labels, each one of the form σ/τ, where $\sigma \in A_{in} = (env) \times (\mathcal{O}.\Sigma_{in})$ and $\tau \in A^*_{out} = ((\mathcal{O}) \times (\mathcal{O}.\Sigma_{out}))^*$. Let $\eta = a^0 \cdot a^1 \cdots$ where $a^i \in A_{in}$. The **trace set** of A on η is the language $\mathcal{L}^\eta_A \subseteq (A^* \cup A^\omega)$, such that a word $w = w_0 \cdot w_1 \cdot w_2 \cdots$ is in $\mathcal{L}^\eta_A$ iff $w_i = a^i \cdot x^i$, $x^i = x^{i_0} \cdots x^{i_{k_i - 1}} \in A^*_{out}$ and there exists a sequence of states $q^{0_1}, q^{1_1}, ..., q^{1_{k_1}}, q^{2_1}, ..., q^{2_{k_2}}, ...$ with $q^{0_1} = q_0$, and such that for all i, j $(q^{i_j}, /x^{i-1_j}, q^{i_{j+1}}) \in \delta$ and $(q^{i_{k_i}}, a^i/x^{i_0}, q^{i+1_1}) \in \delta$.

The satisfaction relation between a GSA and an LSC specification is defined as for object systems: the GSA A satisfies $LS = \langle M, amsg, mod \rangle$, written $A \models LS$, if $\forall m \in M, mod(m) = universal \Rightarrow \forall \eta\ \mathcal{L}^\eta_A \subseteq \mathcal{L}_m$, and $\forall m \in M$, $mod(m) = existential \Rightarrow \exists \eta\ \mathcal{L}^\eta_A \cap \mathcal{L}_m \neq \emptyset$.

Since LS is consistent, there exists a language $\mathcal{L}_1$ as in Def. 2. Since $\mathcal{L}_1$ is regular, there exists a DFA $\mathcal{A} = (A, S, s_0, \rho, F)$ accepting it. We may assume that $\mathcal{A}$ is minimal, so all states in S are reachable and each state leads to some accepting state.

From Clause 2 of Def. 2, for every accepting state s of $\mathcal{A}$ and for every $a \in A_{in}$ there exists an outgoing transition with label a leading to a state that is connected to an accepting state by a path labeled $r \in A^*_{out}$. Formally,

$$\forall s \in F\ \forall a \in A_{in}\ \ \rho(s, a) = s'\ \Rightarrow\ \exists r \in A^*_{out} \text{ s.t. } \rho(s', r) \in F$$

From Clause 3 of Def. 2, no nonaccepting states of $\mathcal{A}$ have any outgoing transitions with label $a \in A_{in}$. This is true since if there were such a state $s \notin F$ reachable from the initial state by x, we would have $\rho(s_0, x) = s$ and $\rho(s, a) = s'$, and from s' we can reach an accepting state $\rho(s', z) \in F$. Then $w = x \cdot a \cdot z$ would violate Clause 3.

We have shown that $\mathcal{A}$ has transitions labeled by A_{in} only for accepting states, and for an accepting state there is such a transition for every letter from A_{in}. We now convert $\mathcal{A}$ into an NFA $\mathcal{A}'$ with the same properties, but, in addition, accepting states do not have outgoing transitions labeled A_{out}. This can be done by adding, for each state $s \in F$, an additional state $s' \notin F$. All incoming transitions into s are duplicated so that they also enter s' and all outgoing transitions from s labeled A_{out} are transferred to s'. $\mathcal{A}'$ accepts the same language as $\mathcal{A}$ since it can use nondeterminism to decide if to take a transition to s or s'.

We now transform the automaton $\mathcal{A}'$ into a GSA $\mathcal{B}$ by changing all transitions with a label from A_{out} into null transitions with that letter as an action. All transitions with a label from A_{in} are left unchanged.

We have to show that $\mathcal{B}$ satisfies the specification LS. From the construction of $\mathcal{B}$, we have

$$\mathcal{L}_B = \cup_\eta \mathcal{L}^\eta_B = \mathcal{L}_1$$

From Clause 1 of Def. 2, we have

$$\mathcal{L}_1 \subseteq \bigcap_{m_j \in M, mod(m_j)=universal} \mathcal{L}_{m_j}$$

Hence,

$$\forall m \in M, mod(m) = universal \Rightarrow \mathcal{L}_1 \subseteq \mathcal{L}_m,$$

yielding

$$\forall m \in M, mod(m) = universal \Rightarrow \forall \eta\ \mathcal{L}_B^\eta \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_m$$

This proves Clause 1 of Def. 1.
Now, from Clause 4 of Def. 2, we have

$$\forall m \in M, mod(m) = existential \Rightarrow \mathcal{L}_m \cap \mathcal{L}_1 \neq \emptyset$$

But since $\mathcal{L}_1 = \cup_\eta \mathcal{L}_B^\eta$, this becomes

$$\forall m \in M, mod(m) = existential \Rightarrow \exists \eta\ \mathcal{L}_B^\eta \cap \mathcal{L}_m \neq \emptyset,$$

thus proving Clause 2 of Def. 1.

Applications of Finite-State Transducers in Natural Language Processing

Lauri Karttunen

Xerox Research Centre Europe,
6, chemin de Maupertuis, F-38240 Meylan, France
karttunen@xrce.xerox.com http://www.xrce.xerox.com

Abstract. This paper is a review of some of the major applications of finite-state transducers in Natural Language Processing ranging from morphological analysis to finite-state parsing. The analysis and generation of inflected word forms can be performed efficiently by means of lexical transducers. Such transducers can be compiled using an extended regular expression calculus with restriction and replacement operators. These operators facilitate the description of complex linguistic phenomena involving morphological alternations and syntactic patterns. Because regular languages and relations can be encoded as finite-automata, new languages and relations can be derived from them directly by the finite-state calculus. This is a fundamental advantage over higher-level linguistic formalisms.

1 Introduction

The last decade has seen a substantial surge in the use of finite-state methods in many areas of natural language processing. This is a remarkable comeback considering that in the dawn of modern linguistics, finite-state grammars were dismissed as fundamentally inadequate. Noam Chomsky's seminal 1957 work, *Syntactic Structures* [3], includes a short chapter devoted to "finite state Markov processes", devices that we now would call *weighted finite-state automata*. In this section Chomsky demonstrates in a few paragraphs that

> English is not a finite state language. (p. 21)

In any natural language, a sentence may contain discontinuous constituents embedded in the middle of another discontinuous pair as in "If$_1$ ... either$_2$ ... or$_2$ ... then$_1$ ..." It is impossible to construct a finite automaton that keeps track of an unlimited number of such nested dependencies. Any finite-state machine for English will accept strings that are not well-formed.

The persuasiveness of *Syntactic Structures* had the effect that, for many decades to come, computational linguists directed their efforts towards more powerful formalisms. Finite-state automata as well as statistical approaches disappeared from the scene for a long time. Today the situation has changed in a fundamental way: statistical language models are back and so are finite-state

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 34–46, 2001.

automata, in particular, finite-state transducers. One reason is that there is a certain disillusionment with high-level grammar formalisms. Writing large-scale grammars even for well-studied languages such as English turned out to be a very hard task. With easy access to text in electronic form, the lack of robustness and poor coverage became frustrating. But there are other, more positive reasons for reasons for the renewed interest in finite-state techniques. In phonology, it was discovered rather early [6] that the kind formal descriptions of phonological alternations used by linguists were, against all appearances, finite-state models. In syntax, it became evident that although English as a whole is not a finite-state language, there are nevertheless subsets of English for which a finite-state description is not only adequate but also easier to construct than an equivalent phrase-structure grammar. Finally, considerable process has been made in developing special finite-state formalisms that are suited for the description of linguistic phenomena and, along with them, compilers that efficiently produce automata from such a description. The automata in current linguistic applications are typically much too large and complex to be produced by hand.

The following sections will cover these positive developments in more detail.

2 Finite-State Morphology

Morphology is a domain of linguistics that studies the formation of words. It is traditional to distinguish between *surface forms* and their analyses, called *lemmas*. The lemma for a surface form such as the English word `bigger` typically consists of the traditional dictionary citation form of the word together with terms that convey the morphological properties of the particular form. For example, the lemma for `bigger` might be represented as `big+Adj+Comp` to indicate that `bigger` is the comparative form of the adjective `big`.

There are two challenges in modeling natural-language morphology:

1. **Morphotactics**
   Words are typically composed of smaller units of meaning, called morphemes. The morphemes that make up a word must be combined in a certain order: `piti-less-ness` is a word of English but `*piti-ness-less` is not. Most languages build words by concatenation but some languages also exhibit non-concatenative processes such as interdigitation and reduplication [2].
2. **Morphological Alternations**
   The shape of a morpheme often depends on the environment: `pity` is realized as `piti` in the context of less, `die` as `dy` in `dying`.

The basic claim of finite-state approach to morphology is the relation between the surface forms of a language and their corresponding lemmas can be described can be modeled as a *regular relation*.[1] If the relation is regular, it can be defined using the metalanguage of regular expressions; and, with a suitable compiler, the regular expression source code can be compiled into a finite-state transducer that implements the relation computationally.

[1] Some writers prefer the term *rational relation*.

In the resulting transducer, each path (= sequence of states and arcs) from the initial to a final state represents a mapping between a surface form and its lemma, also known as the *lexical form*. For example, the information that the comparative of the adjective `big` is `bigger` might be represented in the English lexical transducer by the path in Figure 1 where the zeros represent epsilon symbols.[2]

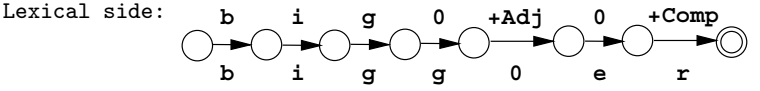


Fig. 1. A Path in a Transducer for English

For the sake of clarity, Figure 1 represents the upper and the lower side of the arc label separately on the opposite sides of the arc. In the rest of the paper, we use a more compact notation: the upper and the lower symbol are combined into a single label of the form `upper:lower` if the symbols are distinct. Identity pairs, e.g. `b:b`, are reduced to a single symbol. In standard notation, the path in Figure 1 is labeled as

`b i g 0:g +Adj:0 0:e +Comp:r`.

An important characteristic of the finite-state transducers built at Xerox is that they are inherently bidirectional: there is no privileged input side. The path in Figure 1 can be traversed matching either the form `bigger` to produce `big+Adj+Comp`, or vice versa. The same transducer can be used for analysis (surface input, "upward" application) or for generation (lexical input, "downward" application).

A single surface string can be related to multiple lexical strings. For example, a morphological transducer for French applied upward to the surface string `suis` may produce the four lexical strings shown in Figure 2. Ambiguity in the downward direction is also possible, as in the relation of the lexical string `payer+IndP+SG+P1+Verb` ("I pay") to the surface strings `paie` and `paye`, which are in fact alternate spellings in standard French orthography.

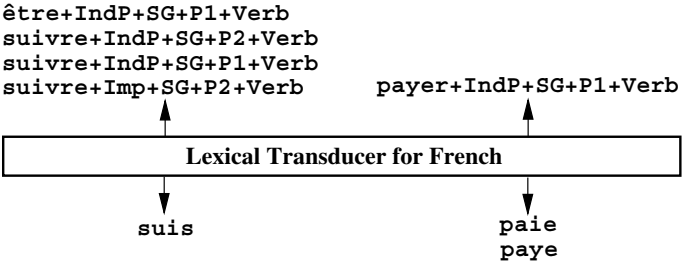


Fig. 2. Morphological Ambiguities

[2] The epsilon symbols and their placement in the string is not significant. We will ignore them whenever it is convenient.

At Xerox, such lexical transducers have been created for a great number of languages including most of the European languages, Turkish, Arabic, Korean, and Japanese. The source descriptions are written using notations [12,9, 1] that are helpful shorthands for ordinary regular expressions. The construction is commonly done by creating two separate modules: a lexicon description that defines the morphotactics of the language and a set of rules that define regular alternations such as the gemination of `g` and the epenthetical `e` in the surface form `bigger`. Irregular alternations such as `être:suis` are defined explicitly in the source lexicon. The separately compiled lexicon and rule networks are subsequently composed together as in Figure 3.

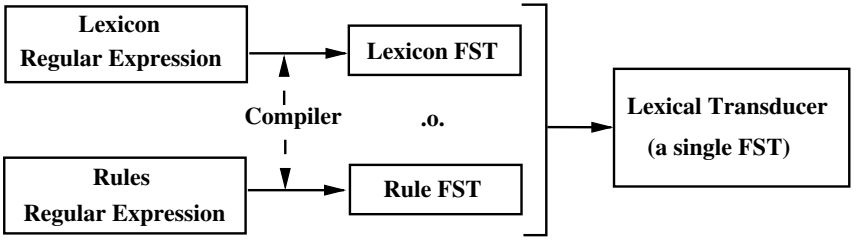


Fig. 3. Creation of a Lexical Transducer

Lexical transducers may contain hundreds of thousands, even millions, of states and arcs and an infinite number of paths in the case of languages such as German that in principle allow noun compounds of any length. The regular expressions from which such complex networks are compiled include high-level operators that have been developed in order to make it possible to describe constraints and alternations that commonly found in natural languages in a convenient and perspicuous way. We will describe them in the following sections.

3 Basic Expression Calculus

The notation used in this section comes from the Xerox finite-state calculus. It is described in detail in Chapter 2 of the forthcoming book by Beesley and Karttunen [1]. We use uppercase letters, `A`, `B`, etc., as variables over regular expressions. Lower case letters, `a`, `b`, etc., stand for symbols. There are two special symbols: **0**, the EPSILON symbol, that stands for the empty string and **?**, the ANY symbol that represents the infinite set of symbols in some yet unknown alphabet. The special meaning of `0`, `?`, and any other symbol can be canceled by enclosing the symbol in double quotes.

An atomic expression consisting of a symbol pair such as `a:b` denotes a relation containing the corresponding strings. An expression consisting of a single symbol such as `a` denotes the language consisting of "a" or, alternatively, the corresponding identity relation. The Xerox implementation intentionally does not distinguish between `a` and `a:a`.

Complex regular expressions can be built up from simpler ones by means of regular expression operators. Square brackets, `[]`, are used for grouping expressions. Because both regular languages and regular relations are closed under concatenation and union, the following basic operators can be combined with any kind of regular expression:

`A | B`	Union.
`A B`	Concatenation.
`(A)`	Optionality; equivalent to `[A | 0]`.
`A+`	Iteration; one or more concatenations of `A`.
`A*`	Kleene star; equivalent to `(A+)`.

Although regular languages are closed under complementation and intersection, regular relations are not [8]; thus the following operators can be combined only with expressions that denote a regular language.

`~A`	Complement
`\A`	Term complement; all single symbol strings not in `A`.
`A & B`	Intersection
`A - B`	Subtraction (minus)

Regular relations can be constructed by means of two basic operators:

`A .x. B`	Crossproduct
`A .o. B`	Composition

The crossproduct operator, `.x.`, is used only with expressions that denote a regular language; it constructs a relation between them. `[A .x. B]` designates the relation that maps every string of `A` to every string of `B`.

4 Containment, Restriction, Replacement, and Marking

The syntax (though not the descriptive power) of regular expressions can be extended by defining new operators that allow commonly used constructions to be expressed more concisely. A simple example of a trivial but convenient extension is the CONTAINMENT operator `$`.

`$A` $=_{def}$ `[?* A ?*]`

For example, `$[a | b]` denotes all strings that contain at least one "a" or "b" somewhere.

The addition of new operators can be more than just a notational convenience. A case in point is Koskenniemi's [16] RESTRICTION operator `=>`.

`A => L _ R`	Restriction; `A` only in the context of `L _ R`.

Here A, L and R may denote any regular language. This expression designates the language of strings that have the property that any string of A that occurs in them is immediately preceded by some string from L and immediately followed by some string from R. For example, `a => b _ c` includes all strings that contain no occurrence of "a", strings like "bac-bac" that completely satisfy the condition, but no strings like "ab". A special boundary symbol, `.#.`, is used to indicate the beginning or the end of the string. For example, `a => _ .#.` allows "a" only at the end of a string.

The advantage of the restriction operator is that it encodes in a compact way a useful condition that is difficult to express in terms of the more primitive operators. The definition of `[A => L _ R]` is shown below.

`A => L _ R` $=_{def}$ `[~[ [~[?* L] A ?*] | [?* A ~[R ?*]] ]]`

Another example of a useful high-level abstraction is the REPLACE operator `->`. As we will see shortly, there are many constructions involving this operator. The simplest variant is unconstrained, obligatory replacement:

`A -> B` Replacement of A by B.

Transducers compiled from `->` expressions are usually intended to be applied downward; they can of course be inverted and applied in the other direction. The component expressions, A and B, must denote regular languages but the expression as a whole denotes a relation. The `[A -> B]` relation maps any upper-language string to itself if the string contains no instance of A. Upper language strings that contain instances of A are paired with lower-language strings that are otherwise identical except that each A segment is replaced by some B string. The definition [10] of simple replacement is shown below.

`A -> B` $=_{def}$ `[ [~$[A - 0] [A .x. B]]* ~$[A - 0]]`

Two replace expressions linked with a comma indicate parallel replacement. For example,

```
a -> b, b -> a
```

yields a transducer that exchanges the two letters mapping "abba" to "baab".

High-level abstractions like `A => L _ R` and `A -> B` are conceptually easier to operate with than the logically equivalent but very complex primitive formulas, just as it is easier to write complex computer programs in a high-level language rather than in a logically equivalent assembly language.

Instead of replacing the strings of a language by other strings, it is sometimes useful just to mark them in some special way. In the Xerox calculus, an expression of the form

`A -> B ... C` Marking A by B and C.

yields a transducer that maps any upper language string to a lower-language string that is identical to it except that any instance of `A` is preceded by a string from `B` and followed by a string from `C`. Here `A`, `B` and `C` may denote any regular language. In practice, however, `B` and `C` are usually atomic expressions. For example, `a | e | i | o | u -> "[" ... "]"` yields a transducer that encloses vowels between square brackets leaving the rest of the text unchanged. The relation includes pairs such as

```
 i c e c r e  a m
[i]c[e]c r[e][a]m
```

4.1 Constraining Replacement and Marking

Replacement and marking can be constrained in many different ways: by context, by direction of application, by length of the pattern to be replaced or marked. The basic technique for compiling constrained replacement and marking transducers was invented in the early 1980's by Kaplan and Kay [7] for Chomsky-Halle-type rewrite rules [4]. It was also used very early for Koskenniemi's two-level rules [16,14,12]. The idea was finally explained in detail in Kaplan and Kay's 1994 paper [8]. There is now a rich literature on this topic [10,17,5,11,15,18]. The details vary but the basic method involves composing together a cascade of networks that introduce various auxiliary symbols into the input string, constrain their distribution, and finally eliminate the auxiliary alphabet. As there is no space to explore the compilation issue in a technical way, we will only explain the syntax of constrained replacement and marking expressions and give a few examples of the corresponding transducers without explaining how the expressions are compiled.

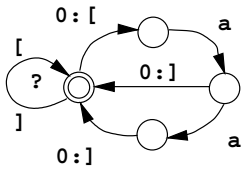
The transducers compiled from the simple replacement and marking expressions are in general ambiguous in the sense that a string in the upper language of the relation is paired with more than one lower-language string. For example, `a | a a -> "[" ... "]"` yields a marking transducer than maps the upper language string "aaa" into three different lower-language strings:

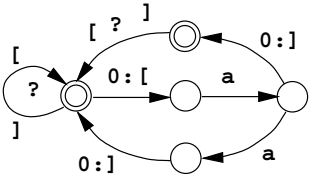
```
 a  a  a      a  a a      a a  a
 -  -  -      -  ---      ---  -
[a][a][a]    [a][a a]    [a a][a]
```

The `->` operator does not constrain the selection of the alternate substrings for replacement or marking. In this case, the upper language string can be factored or parsed in three different ways.

For many applications, it is useful to define another version of replacement and marking that in all such cases yields a unique outcome. The longest-match, left-to-right replace operator, `@->`, defined in [11], imposes a unique factorization on every input. The upper language substrings to be marked or replaced are selected from left to right, not allowing any overlaps. If there are alternate candidate strings starting at the same location, only the longest one is chosen. Thus `a | a a @-> "[" ... "]"` denotes a relation that unambiguously maps

"aaa" to "[aa][a]". The transducers corresponding to the `->` and `@->` variant of this expressions are shown in Figure 4.[3]






```
a | a a -> "[" ... "]"          a | a a @-> "[" ... "]"
```

Fig. 4. An Ambiguous and an Unambiguous Marking Transducer

Replacement and marking contexts can be specified using same notation as for restriction: `L _ R`, where `L` is the left context, `R` is the right context, and `_` marks the site of the upper language string that is replaced or marked. In the case of a restriction expression, the interpretation of context is self-evident because a restriction denotes a set of strings. This is not the case for replacement and marking. Replacement and marking expressions must specify whether `L` and `R` pertain to the upper or the lower side of the relation. The Xerox calculus provides specific markers `||`, `//`, \\ and \/ to distinguish between the four possible cases:

`|| L _ R`	L and R both on the upper side
`// L _ R`	L on the lower, R on the upper side
`\\ L _ R`	L on the upper, R on the lower side
`\/ L _ R`	L and R both on the lower side

To see the difference between, say `||` and `//`, versions let us consider two variants of a phonological rule that shortens a double "aa" in the context of another double "aa" in the preceding syllable. Here `C` represents any consonant.

Rule 1.	`a a -> a || a a C+ _`	(Slovak)
Rule 2.	`a a -> a // a a C+ _`	(Gidabal)

Vowel shortening is a very common type of morphological alternation under many different kinds of context conditions. Interestingly, in some languages such as Slovak the shortening depends on the lexical (upper side) context whereas in languages such as Gidabal (an Australian language), it is conditioned by the surface side.[4] The hypothetical lexical form "baacaadaafaa" would be realized quite differently in these two languages:

```
b a a c a a d a a f a a          b a a c a a d a a f a a
b a a c a   d a   f a            b a a c a   d a a f a
        Rule 1                           Rule 2
```

[3] The symbol **?** in an arc label represents an UNKNOWN symbol; in this case, any symbol other than `[`, `]`, and `a`. By convention, the leftmost state is the start state, final states are indicated by double circles.

[4] This example is due to Martin Kay (p.c.).

In a language like Slovak, the last three syllables would all shorten yielding "baacadafa" whereas a language like Gidabal would show the alternating pattern "baacadaafa".

The two replacement transducers compiled from Rule 1 and Rule 2 are shown in Figure 5.

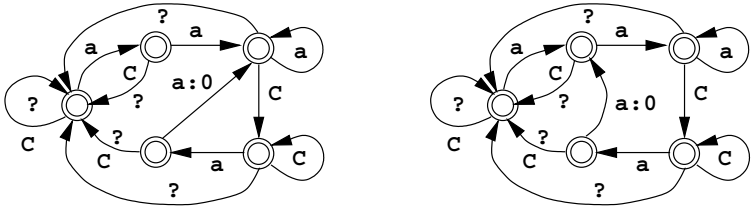


Fig. 5. Two Vowel-Shortening Rules

Contextual constraints may be combined with the directional left-to-right and longest match constraints. For example, if `C` and `V` stand for consonants and vowels, respectively, a simple syllabification rule may be expressed in the following way:

```
C* V+ C* @-> ... "-" || _ C V
```

This marking expression yields an unambiguous transducer that inserts a hyphen after each longest available instance of the `C* V+ C*` pattern that is followed by a consonant and vowel. The relation it encodes consists of pairs of strings such as

```
s t r u k   t u   r a   l i s   m i
s t r u k - t u - r a - l i s - m i          .
```

In this case, the choice between `||` and `//` makes no difference but the two other context markers, `\\` and `\/` could not be used here.

The syllabification transducer is a simple finite-state parser: it recognizes and marks instances of a regular language in a text. In the next section we will show a more sophisticated example of this kind.

5 Finite-State Syntax

Although the syntax of a natural language cannot in general be described by a finite-state, or even a context-free grammar there are many subsets of natural language that can be correctly described by very simple means, for example, names and titles, addresses, prices, dates, etc. In this section, we examine one such case in detail: a grammar for dates.

For the sake of illustration, let us consider here only one of several common date formats, expressions such as

```
                    Tuesday
July 25             Tuesday, July 25
July 25, 2000       Tuesday, July 25, 2000
```

In the following we assume that a date expression consists of a day of the week, a month and a date with or without a year, or a combination of the two. Note that this description of the syntax of date expressions presents the same problem we encountered in the `a | aa @-> a` example in the previous section. Long date expressions, such as "Tuesday, July 25, 200", contain smaller well-formed date expressions, e.g. "July 25", that should be ignored in the context of a larger date. In order to simplify the presentation, we stipulate that date expressions are contiguous strings, including the internal spaces and commas.

To facilitate the specification of the date language we first define some auxiliary terms and then use them to define a language of dates and a parser for the language. The complete set of definitions is shown below:

```
1To9  =  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
0To9  =  "0" | 1To9
Day   =  Monday | Tuesday | ..... | Saturday | Sunday
Month =  January | February | ..... | November | December
Date  =  1To9 | [1 | 2] 0To9 | 3 ["0" | 1]
Year  =  1To9 (0To9 (0To9 (0To9)))
AllDates = Day | (Day ", ") Month " " Date (", " Year)
```

From these definitions we can compile a small finite-state automaton, `AllDates`, with 13 states and 96 arcs that describes a language of about 30 million date expressions for the period from January 1, 1 to December 31, 9999.

A parser for the language can be compiled from the following simple regular expression.

```
AllDates @-> "[" ... "]"
```

It yields a transducer of 23 states and 321 arcs that marks maximal date expressions in the manner illustrated by the following text:

```
Today is [Tuesday, July 25, 2000] because yesterday was [Monday]
and it was [July 24] so tomorrow must be [Wednesday, July 26].
```

Because of the left-to-right, longest-match constraints associated with the `@->` operator, the transducer brackets only the maximal date expressions.

However, this regular-expression grammar is not optimal. The `AllDates` language includes a large number of syntactically correct but semantically invalid date expressions. For example, there is no "April 31, 2000", "February 29, 1900", or "Sunday, September 29, 1941". April only as 30 days in any year; unlike year 2000, year 1900 was not a leap year; and September 29, 1941 fell on a Monday.

All these three types of imperfections can be corrected within the finite-state calculus. For each of these three types on invalid dates we can define a regular language that excludes such expressions. By intersecting these constraint languages with the `AllDates` language, we can define a language that contains only semantically valid dates. Figure 6 illustrates the idea.

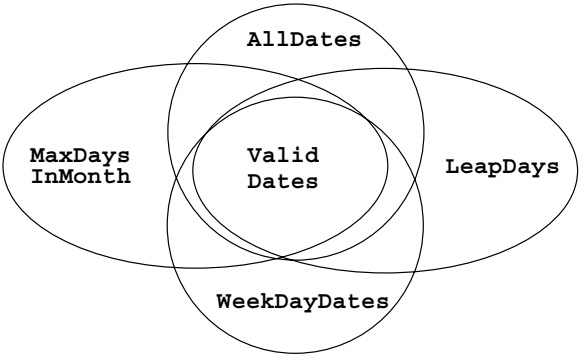


Fig. 6. Refinement by Intersection

We need three constraints:

`MaxDaysInMonth`	Restriction on the distribution of 30 and 31.
`LeapDays`	Restriction on February 29.
`WeakDayDate`	Restrictions on weekdays and dates

In fact, all the constraints can be expressed by means of the restriction operator `=>` defined in the previous section. For example, to build the leap day constraint we first need to define the language of leap years, that is the language of all numbers divisible by four but subtracting centuries such as 1900 that are not divisible by 400.

```
Even =      "0" | 2 | 4 | 6 | 8
Odd =       1 | 3 | 5 | 7 | 9
N =         1To9 0To9*
Div4 =      [((N) Even) ["0" | 4 | 8]] | [(N) Odd [2 | 6]]
LeapYears = Div4 - [[N - Div4] "0" "0"]
```

Here we first define `Div4` as the infinite set of natural numbers that are divisible by four. This set consists of two parts: numbers that end in 0, 4, or 8 possibly preceded by an even number and numbers that end in 2 or 6 preceded by an odd number. Finally, we define `LeapYears` as the set of numbers divisible by 4 subtracting centuries that are not multiples of 400. Note that the expression `[N - Div4] "0" "0"` denotes numbers with two final zeros that are preceded by a number that is not divisible by four. For example, it includes "1900" but not "2000". Because `LeapYears` is defined as `Div4` minus this set, it follows that the string "2000" is in the language but "1900" is not.

Once the language of leap years is defined, the distribution of "February 29" in date expressions can be constrained with the following simple restriction.

```
LeapDays = February " " 2 9 ", "  => _ LeapYears .#.
```

In other words: a date expression containing "February 29, " must terminate with a leap year. The boundary symbol, `.#.`, is necessary here to mark the end of the year string in order to rule out expressions like "February 29, 1969" which

would qualify if we were allowed to take into account only the first three digits since year 196 is a leap year in the Gregorian calendar.

The construction of the `WeakDayDate` constraint is not as trivial but not as difficult as it might initially seem. See [13] for details. Having constructed the auxiliary constraint languages we can define the language of valid dates as

```
ValidDates = AllDates & MaxDaysInMonth & LeapDays & WeekDayDates
```

The network contains 805 states, 6472 arcs, and about 7 million date expressions.

We could now construct a parser that recognizes only valid dates. But we actually can do something more interesting, namely, define a parser that recognizes all date expressions and marks them as valid, "[VD", or invalid, "[ID":

```
              ValidDates] @-> "[VD" ... "]" ,
[AllDates - ValidDates] @-> "[ID" ... "]"
```

This parallel replacement expression compiles into a 2699 state, 20439 arc transducer in about 15 seconds on a Sun workstation. The time includes the compilation of all the auxiliary expressions and constraints discussed above. The following example illustrates the effect of the transducer on a sample text.

```
The correct date for today is [VD Tuesday, July 25, 2000].
Today is not [ID Tuesday, July 26, 2000].
```

6 Conclusion

Although regular expressions and the algorithms for converting them into finite-state automata have been part of elementary computer science for decades, the restriction, replacement, and marking expressions we have focused on are relatively recent. They have turned out to be very useful for linguistic applications in particular for morphology, tokenization, and shallow parsing. Descriptions consisting of regular expressions can be efficiently compiled into finite-state networks, which in turn can be determinized, minimized, sequentialized, compressed, and optimized in other ways to reduce the size of the network or to increase the application speed. Many years of engineering effort have produced efficient runtime algorithms for applying networks to strings.

Regular expressions have a clean, declarative semantics. At the same time they constitute a kind of high-level programming language for manipulating strings, languages, and relations. Although regular grammars can cover only limited subsets of a natural language, there can be an important practical advantage in describing such sublanguages by means of regular expressions rather than by some more powerful formalism. Because regular languages and relations can be encoded as finite automata, they can be more easily manipulated than context-free and more complex languages. With regular expression operators, new regular languages and relations can be derived directly without rewriting the grammars for the sets that are being modified. This is a fundamental advantage over higher-level formalisms.

References

1. Kenneth R. Beesley and Lauri Karttunen. *Finite-State Morphology: Xerox Tools and Techniques*. Cambridge University Press, 2000. To appear.
2. Kenneth R. Beesley and Lauri Karttunen. Finite-state non-concatenative morphotactics. In Lauri Karttunen Jason Eisner and Alain Thériault, editors, *SIGPHON-2000*, pages 1–12, August 6 2000. Proceedings of the Fifth Workshop of the ACL Special Interest Group in Computational Phonology.
3. N. Chomsky. *Syntactic Structures*. Mouton, Gravenhage, Netherlands, 1957.
4. Noam Chomsky and Morris Halle. *The Sound Pattern of English*. Harper and Row, New York, 1968.
5. Edmund Grimley-Evans, George Anton Kiraz, and Stephen G. Pulman. Compiling a partition-based two-level formalism. In *Proceedings of the 16th International Conference on Computational Linguistics*, Copenhagen, 1996.
6. C. Douglas Johnson. *Formal Aspects of Phonological Description*. Mouton, The Hague, 1972.
7. Ronald M. Kaplan and Martin Kay. Phonological rules and finite-state transducers. In *Linguistic Society of America Meeting Handbook, Fifty-Sixth Annual Meeting*, New York, December 27-30 1981. Abstract.
8. Ronald M. Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.
9. Lauri Karttunen. Finite-state lexicon compiler. Technical Report ISTL-NLTT-1993-04-02, Xerox Palo Alto Research Center, Palo Alto, CA, April 1993.
10. Lauri Karttunen. The replace operator. In *ACL'95*, Cambridge, MA, 1995. cmp-lg/9504032.
11. Lauri Karttunen. Directed replacement. In *ACL'96*, Santa Cruz, CA, 1996. cmp-lg/9606029.
12. Lauri Karttunen and Kenneth R. Beesley. Two-level rule compiler. Technical Report ISTL-92-2, Xerox Palo Alto Research Center, Palo Alto, CA, October 1992.
13. Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Journal of Natural Language Engineering*, 2(4):305–328, 1996.
14. Lauri Karttunen, Kimmo Koskenniemi, and Ronald M. Kaplan. A compiler for two-level phonological rules. Technical report, Xerox Palo Alto Research Center and Center for the Study of Language and Information, Stanford University, June 25 1987.
15. André Kempe and Lauri Karttunen. Parallel replacement in finite-state calculus. In *COLING'96*, Copenhagen, August 5–9 1996. cmp-lg/9607007.
16. Kimmo Koskenniemi. Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki, 1983.
17. Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *ACL'96*, Santa Cruz, CA, 1996.
18. Gertjan van Noord and Dale Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam Germany, 1999.

Fast Implementations of Automata Computations

Anne Bergeron and Sylvie Hamel

LACIM, Université du Québec à Montréal,
C.P. 8888 Succursale Centre-Ville, Montréal, Québec,
Canada, H3C 3P8,
anne@lacim.uqam.ca

Abstract. In [6], G. Myers describes a bit-vector algorithm to compute the edit distance between strings. The algorithm converts an input sequence to an output sequence in a parallel way, using bit operations readily available in processors.

In this paper, we generalize the technique, and characterize a class of automata for which there exists equivalent parallel, or *vector*, algorithms. As an application, we extend Myers result to arbitrary weighted edit distances, which are currently used to explore the vast data-bases generated by genetic sequencing.

1 Introduction

Finite automaton are powerful devices for computing on sequences of characters. Among the finest examples, very elegant linear algorithms have been developed for the string matching problem [1]. Automata are also widely used in fields such as metric lexical analysis [3] or bio-computing, where *approximate string matching* is at the core of most algorithms that deal with genetic sequences [4]. In these fields, the huge amount of data to be processed – sometimes billions of characters – calls for algorithms that are better than linear.

One way to accelerate the computations is to exploit the parallelism of vector operations, especially bit-vector operations. For example, in [2] and [5], bit-vectors are used to code the set of states of a non-deterministic automaton. In this paper, as in [6], we want to accelerate computations done with deterministic automata, and we use vectors to represent sequences of events or sequences of states.

Given a deterministic finite automaton, and an input sequence $x_1 \dots x_m$, we are interested in the output sequence $y_1 \dots y_m$ of *visited states*. Since executing one transition is usually considered to be a constant time operation, the output sequence can be obtained in $\mathcal{O}(m)$ time.

In order to improve the efficiency of such algorithms, we have to find quicker ways to obtain $y_1 \dots y_m$ from $x_1 \dots x_m$. The best possible algorithm would do it in constant time:

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 47–56, 2001.

$$\begin{array}{c} x_1 x_2 \ldots x_m \\ \downarrow \\ y_1 y_2 \ldots y_m \end{array}$$

Clearly, this can seldom be done when m is unbounded. The next best alternative would be to obtain $y_1 \ldots y_m$ with a bounded number of *vector* operations on $x_1 \ldots x_m$:

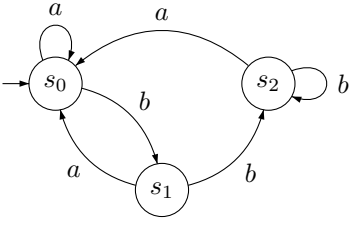
$$\begin{array}{c} x_1\ x_2\ \ldots\ x_m \\ \downarrow\ \downarrow \qquad \downarrow \\ y_1\ y_2\ \ldots\ y_m \end{array}$$

Indeed, vector operations can be implemented in parallel, in dedicated circuits, or using high-speed bit-wise operations available in processors. The drawback is that vector operations are applied component by component, meaning that the only computations that one could hope to solve with *pure* vector operations are those where the value of y_i depends only on the value of x_i, and its close neighbors.

On the bright side, some bit operations widely available in processors do have a *memory* of past events. Using these, it is possible to parallelize complex computations done with automata.

2 The Basics of Vector Algorithms

As a simple example, consider the following automaton. On input sequence *babba*, it will generate the output $s_1 s_0 s_1 s_2 s_0$ – we omit the leading initial state.



Let's associate to a string $x_1 \ldots x_m$, the *characteristic vector* of a letter l, denoted by the bold letter $\boldsymbol{l}$, and defined by:

$$\boldsymbol{l}_i = \begin{cases} 1 \text{ if } x_i = l \\ 0 \text{ otherwise} \end{cases}$$

Characteristic vectors are sequences of bits, and we will operate on them with the standard bit operations: bit-wise logical operators, left and right shifts, binary addition, etc. We can obtain, for example, the characteristic vector of a set S of letters by computing the disjunction of the characteristic vectors of the letters in S.

In our example, we have the following *direct* computation of the characteristic vectors $\boldsymbol{s_0}$, $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$ of the output sequence $y_1 \dots y_m$, in terms of the characteristic vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of the input sequence.

$$\boldsymbol{s_0} = \boldsymbol{a} \tag{1}$$
$$\boldsymbol{s_1} = (\uparrow_1 \boldsymbol{s_0}) \wedge \boldsymbol{b} \tag{2}$$
$$\boldsymbol{s_2} = \neg(\boldsymbol{s_0} \vee \boldsymbol{s_1}) \tag{3}$$

where $\uparrow_b \boldsymbol{l}$ stands for a right shift of the vector $\boldsymbol{l}$ with the value b filled in in the first position.

These three equations are used in the following way. Suppose, for example, that the input sequence given to the automaton is *babba*. The characteristic vectors of the letters a and b are thus:

$$\boldsymbol{a} = 01001$$
$$\boldsymbol{b} = 10110$$

Equation (1) states that the output is s_0 if and only if the input is a, thus:

$$\boldsymbol{s_0} = 01001$$

The output state is s_1 when the input letter is b, and the preceding state is s_0. This can be computed, as in equation (2), by shifting vector $\boldsymbol{s_0}$ to the right and taking the bit-wise conjunction with vector $\boldsymbol{b}$.

$$\begin{aligned} \boldsymbol{s_1} &= 10100 \wedge 10110 \\ &= 10100 \end{aligned}$$

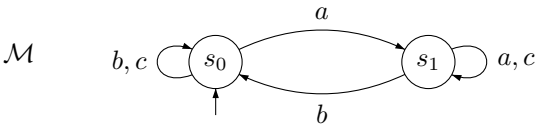
In all other cases, the output state is s_2, which can be expressed, as in equation (3), by the bit-wise negation of the disjunction of characteristic vectors $\boldsymbol{s_0}$ and $\boldsymbol{s_1}$:

$$\begin{aligned} \boldsymbol{s_2} &= \neg(01001 \vee 10100) \\ &= 00010 \end{aligned}$$

If we assume that vector operations are done in parallel then, regardless of the length of the input sequence, the characteristic vectors of the output can be computed with 4 operations! This example is simple, since the output state depends on at most two input letters, but it gives the flavor of the technique. In general, the output states will depend on arbitrarily "far" events but, in some interesting cases, it will still be possible to reduce the computation to direct bit-vector operations.

2.1 Remembering Past Events

The simplest example of a computation that is influenced by past events is given by the following automaton $\mathcal{M}$.

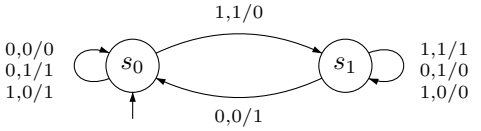


In this case, whether input c yields state s_0 or s_1 depends on events that can be arbitrarily far, as shown by the two input sequences ac^n and bc^n. Fortunately, we have the following formulas that compute $\boldsymbol{s_0}$ and $\boldsymbol{s_1}$ using binary addition with carry propagation – performed from left to right on the bit-vectors:

The Addition Lemma: The characteristic vectors of states $\boldsymbol{s_0}$ and $\boldsymbol{s_1}$ of $\mathcal{M}$ are

$$\boldsymbol{s_0} = \boldsymbol{b} \vee [\boldsymbol{c} \wedge (\neg \boldsymbol{b} + \neg(\boldsymbol{b} \vee \boldsymbol{c}))]$$
$$\boldsymbol{s_1} = \boldsymbol{a} \vee [\boldsymbol{c} \wedge \neg(\boldsymbol{a} + (\boldsymbol{a} \vee \boldsymbol{c}))].$$

Proof: As noted in [6], automaton $\mathcal{M}$ is similar to the classical bit addition Moore automaton:



In this automaton, state s_1 means that the *carry bit* is set to 1. This state is reached when both bits are 1, or when the two bits are different, and their sum is 0. Thus, if two vectors $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ are added, the characteristic vector of state s_1 is

$$(\boldsymbol{x_1} \wedge \boldsymbol{x_2}) \vee [((\neg \boldsymbol{x_1} \wedge \boldsymbol{x_2}) \vee (\boldsymbol{x_1} \wedge \neg \boldsymbol{x_2})) \wedge \neg(\boldsymbol{x_1} + \boldsymbol{x_2})].$$

Using the characteristic vectors associated with events a, b and c of the automaton $\mathcal{M}$, define $\boldsymbol{x_1} = \boldsymbol{a}$ and $\boldsymbol{x_2} = \boldsymbol{a} \vee \boldsymbol{c}$. Then:

$$\begin{aligned} \boldsymbol{a} &= \ \ \boldsymbol{x_1} \wedge \boldsymbol{x_2} \\ \boldsymbol{b} &= \neg \boldsymbol{x_1} \wedge \neg \boldsymbol{x_2} \\ \boldsymbol{c} &= \ \neg \boldsymbol{x_1} \wedge \boldsymbol{x_2} \end{aligned}$$

With these identities, automaton $\mathcal{M}$ becomes a sub-graph of the bit addition automaton. Since $\boldsymbol{x_1} \wedge \neg \boldsymbol{x_2}$ is always false, we get, by substitution, the formula $\boldsymbol{s_1} = \boldsymbol{a} \vee [\boldsymbol{c} \wedge \neg(\boldsymbol{a} + (\boldsymbol{a} \vee \boldsymbol{c}))]$. The formula for $\boldsymbol{s_0}$ is derived in a similar way with $\boldsymbol{x_1} = \neg \boldsymbol{b}$ and $\boldsymbol{x_2} = \neg(\boldsymbol{b} \vee \boldsymbol{c})$. ■

2.2 Solving More Complex Automata

In this section, we establish a sufficient condition for the existence of a vector algorithm for an automaton.

Consider a finite *complete* automaton $\mathcal{A}$ on alphabet of events Σ, with states Q and transition function $Q \times \Sigma \xrightarrow{T} Q$. We say that a state s is *solvable* if, for all events $x \in \Sigma$ that do not loop on s, either all states reach s with x, or none does. Formally:

$$\begin{array}{r} \forall s' \in Q\ T(s', x) = s \quad \text{or} \\ \forall s' \neq s \in Q\ T(s', x) \neq s. \end{array}$$

A solvable state can be *removed* from an automaton in the following sense. Let E_s be the set of events for which $T(s', x) = s$ for all s', and $\mathcal{A} \setminus \{s\}$ be the automaton obtained from $\mathcal{A}$ by removing s, and all its pending arrows. Then if s is solvable, $\mathcal{A} \setminus \{s\}$ is still a complete automaton on the alphabet $\Sigma \setminus E_s$, since $T(s', y) \neq s$ if y is not in E_s.

Definition 1. *An automaton $\mathcal{A}$ is* solvable *if it has one state, or if it has one solvable state s, and $\mathcal{A} \setminus \{s\}$ is solvable.*

Theorem 1. *If an automaton $\mathcal{A}$ is solvable, then there exists a vector algorithm for $\mathcal{A}$.*

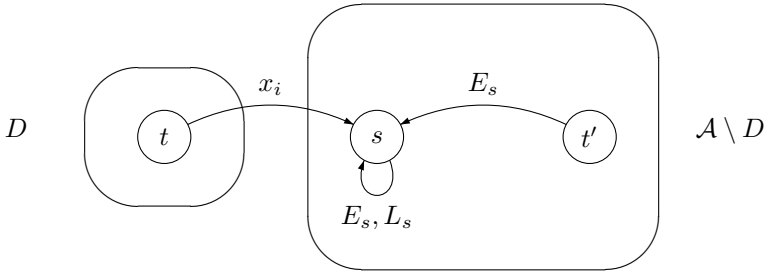
Sketch of Proof. When a state s is solvable, the Addition Lemma can be used to compute the characteristic vector of s. Indeed, let $\boldsymbol{E_s}$ be the characteristic vector of the set E_s, and $\boldsymbol{L_s}$, the characteristic vector of the events that loop on s but are not in E_s. Then:

$$\boldsymbol{s} = \begin{cases} \boldsymbol{E_s} \vee [\boldsymbol{L_s} \wedge (\neg \boldsymbol{E_s} + \neg(\boldsymbol{E_s} \vee \boldsymbol{L_s}))] & \text{or} \\ \boldsymbol{E_s} \vee [\boldsymbol{L_s} \wedge \neg(\boldsymbol{E_s} + (\boldsymbol{E_s} \vee \boldsymbol{L_s}))] \end{cases} \tag{4}$$

according to whether s is initial or not.

Suppose that we have computed the characteristic vectors of a subset D of solvable states, and let $\boldsymbol{K}$ be the disjunction of all those known characteristic vectors.

If s is a solvable state in $\mathcal{A} \setminus D$, we will show that computing the characteristic vector of s is essentially a *local* decision, and can be done in a vectorial way.



The value of $\boldsymbol{s}_i$ will be equal to 1 in three circumstances. First, if the preceding state is known, that is, when $\boldsymbol{K}_{i-1} = 1$, then the value of $\boldsymbol{s}_i = 1$ can be decided

with the transition table of $\mathcal{A}$. If the preceding state is unknown, then it belongs to $\mathcal{A} \setminus D$, and $\boldsymbol{s}_i = 1$ if the input x_i is in E_s. Let $\boldsymbol{N}$ be the characteristic vector resulting from these two possibilities.

Vector $\boldsymbol{N}$ covers, at least, all the cases when s is reached from a different state. In order to account for looping events in L_s, we apply Equation 4 with $\boldsymbol{E_s} = \boldsymbol{N}$. □

Theorem 1 proves the existence of a vector algorithm, but does not give an efficient way to construct one. In the following sections, we will study in details a non trivial application.

3 Approximate String Matching

A common way to formalize the notion of *distance* between two strings is the *edit distance*, based on the number of operations required to transform one string into another. Three basic operations are permitted on individual characters: insertion, deletion and replacement.

For example, in order to transform the string `COMPUTER` into the string `SOLUTION`, we can apply to the first string the sequence of edit operations $RMDRMMIRR$, where R denotes a replacement, D a deletion, I an insertion and M a match.

Such a transformation is usually displayed as an *alignment* of the two strings, where matched or replaced letters are on top of each other, and insertions and deletions are denoted by a properly placed dashes. With our example, we get the alignment:

```
C O M P U T - E R
S O - L U T I O N
```

The *edit distance* between two strings is defined as the minimum number of edit operations – excluding matches – needed to transform one string into another.

A crucial generalization of the edit distance for applications in biology is the *weighted edit distance*. It comes from the observation – in biological sequences – that replacements are not equally likely. Assigning different costs to different edit operations allows the construction of alignments that are meaningful from an evolutionary point of view.

Let c be the cost associated to an insertion or a deletion, and $\delta(a, b)$ be the cost of replacing a by b. We define the cost of a sequence of edit operations to be the sum of the costs of the operations involved. Since a replacement can be achieved by a deletion followed by an insertion, the replacement cost $\delta(a, b)$ should be less than $2c$.

Definition 2. *The* weighted edit distance $\delta(A, B)$ *between two string A and B is the minimal cost to transform A into B.*

In the sequel, we will focus on the following problem. Given a query sequence $P = p_1 \dots p_m$, and a text $T = t_1 \dots t_n$, we want to find the *approximate occurrences* of P in T. Formally, the problem is to find all positions j in T such that, for a given threshold $t \geq 0$, we have $\min_g \delta(P, T[g, j]) \leq t$. Typically, P will be relatively short – a few hundred characters –, while T can be quite large.

The classic solution [8] is obtained by computing the matrix $C[0..m, 0..n]$ with the recurrence relation:

$$C[i,j] = \min \begin{cases} C[i-1, j-1] + \delta(p_i, t_j) \\ C[i, j-1] + c \\ C[i-1, j] + c \end{cases} \tag{5}$$

and initial conditions $C[0, j] = 0$ and $C[i, 0] = ic$.

The successive values of $C[m, j]$ give the desired distances and can be compared to the threshold t. For example, suppose one wants to compute the approximate occurrences of $TATA$ in the text $ACGTAATAGC\dots$ with the usual edit distance, that is $c = 1$ and $\delta(a, b) = 1$ if $a \neq b$. The computation is done with the help of a grid whose cells hold the values of $C[i, j]$. The following table gives a snapshot of the evolving computation:

		A	C	G	T	A	A	T	A	G	C	...
	0	0	0	0	0	0	0	0	0	0	0	...
T	1	1	1	1	0	1	1	0	1	1	1	...
A	2	1	2	2	1	0	1	1	0	1	.....	
T	3	2	2	3	2	1	1	1	1	1	.....	
Line m A	4	3	3	3	3	2	**1**	2	**1**	2	.....	
							↑		↑			

In this table, we can see that, in two occasions, the query string is at one unit of distance from substrings in the text – the substring TAA, at position 6, and the substrings ATA, $AATA$, and $TAATA$, at position 8.

The whole computation can be carried out column by column, requiring $\mathcal{O}(nm)$ time and $\mathcal{O}(m)$ space. In order to do better, we first state a useful lemma that bounds the absolute value of differences between horizontal and vertical values in the matrix:

Lemma 1. *$|C[i, j] - C[i-1, j]| \leq c$ and $|C[i, j] - C[i, j-1]| \leq c$.*

Since the horizontal and vertical differences are bounded, we can code this computation as an automaton, which will turn out to be solvable.

3.1 Computing Distances with an Automaton

With the notations of Equation 5, define:

$$\begin{aligned} \Delta v_{i,j} &= c - (C[i, j] - C[i-1, j]) \\ \Delta h_{i,j} &= C[i, j] - C[i, j-1] + c \end{aligned}$$

From Lemma 1, $\Delta v_{i,j}$ and $\Delta h_{i,j}$ are in the interval $[0..2c]$, and if the successive values of $\Delta h_{m,j}$ are known, then the value of the *score*, $C[m,j]$, can be computed by the recurrence, $C[m,j] = C[m,j-1] + \Delta h_{m,j} - c$, and initial condition $C[m,o] = mc$.

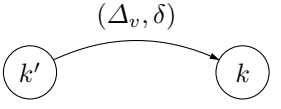
With elementary arithmetic manipulations, Equation 5 translates as:

$$\Delta h_{i,j} = \min \begin{cases} \Delta v_{i,j-1} + \delta(p_i, t_j) \\ \Delta v_{i,j-1} + \Delta h_{i-1,j} \\ 2c \end{cases} \tag{6}$$

with initial conditions $\Delta h_{0,j} = c$ and $\Delta v_{i,0} = 0$. We can thus define an automaton $\mathcal{B}$ that will compute the sequence $\boldsymbol{\Delta h_j} = \Delta h_{1,j} \ldots \Delta h_{m,j}$ given the sequence of pairs:

$$\boldsymbol{(\Delta v_{j-1}, \delta(p, t_j))} = ((\Delta v_{1,j-1}, \delta(p_1, t_j)) \ldots (\Delta v_{m,j-1}, \delta(p_m, t_j)).$$

The states of $\mathcal{B}$ are $\{0, \ldots, 2c\}$, with initial state c, and the transition function of $\mathcal{B}$ is given by following diagram, for an event $(\Delta v, \delta)$ in the cartesian product $[0..2c] \times [0..2c-1]$.



$$\text{where } k = \min \begin{cases} \Delta v + \delta \\ \Delta v + k' \\ 2c. \end{cases}$$

Theorem 2. *Automaton $\mathcal{B}$ is solvable.*

Sketch of proof. We will show that, for $k \in [0..2c)$, k is solvable in $\mathcal{B}_{k-1} = \mathcal{B} \setminus \{0, \ldots, k-1\}$, with the set of events $(\Delta v, \delta)$ such that $\Delta v + \delta = k$.

First note that the only remaining events in $\mathcal{B}_{k-1}$ are those that satisfy $\Delta v + \delta \geq k$. If $\Delta v + \delta = k$, then $\min(\Delta v + \delta, \Delta v + k', 2c) = k$, since $k' \geq k$ for states in $\mathcal{B}_{k-1}$. If both $\Delta v + \delta > k$ and $k' > k$, then the minimum is certainly greater than k. □

Theorem 1 can then be used to produce a corresponding vector algorithm. Note that, in this case, looping events are easily detected, since if $k = k' < 2c$, then either $\Delta v = 0$ or $\Delta v + \delta = k$.

In order to complete the presentation of a vector algorithm for the computation of $\boldsymbol{\Delta v_j}$, we use the relation $\Delta v_{i,j} = \Delta h_{i-1,j} + \Delta v_{i,j-1} - \Delta h_{ij}$ which leads to the vector equation:

$$\boldsymbol{\Delta v_j} = \uparrow_c \boldsymbol{\Delta h_j} + \boldsymbol{\Delta v_{j-1}} - \boldsymbol{\Delta h_j}.$$

4 A Bit-Vector Algorithm

This section contains brief notes on how to implement a *bit*-vector algorithm for the approximate string matching problem.

The first problem is to represent vectors of integers that are not necessarily 0 or 1, but in the bounded interval $[0..2c]$. This can be done easily with an $l \times m$ bit-matrix, where $l = \log(2c) + 1$. There are well known algorithms for all the basic operations – assignment, comparison and arithmetic – on these bit-matrices. These operations are identified with arrows, and bold constants stand for constant vectors.

Assuming that $\boldsymbol{\Delta v}$ is known, the next letter of the text is read, and $\boldsymbol{\delta(p, t_j)}$ is looked up in a pre-computed table. Three vectors, corresponding respectively to the three cases of the proof of Theorem 1, are initialized in the following way:

$\boldsymbol{Sum_1} \leftarrow \boldsymbol{\Delta v}$ % $\boldsymbol{Sum_1}$ will hold the values of $\Delta h_{i-1} + \Delta v_i$.
$\boldsymbol{Sum_2} \leftarrow \boldsymbol{\Delta v} + \boldsymbol{\delta}$ % $\boldsymbol{Sum_2}$ is used to test if $(\Delta v, \delta) \in E_k$.
$\boldsymbol{Loop} \leftarrow (\boldsymbol{\Delta v} = \boldsymbol{0})$ % $\boldsymbol{Loop}$ contains looping events not in E_k.

The characteristic vector $\boldsymbol{N}$ of state k, from 0 to $2c - 1$, is computed with the following equations, keeping track of the known states $\boldsymbol{K}$.

$\boldsymbol{N} \leftarrow (\uparrow_b \boldsymbol{K} \wedge (\boldsymbol{Sum_1} = \boldsymbol{k})) \vee (\boldsymbol{Sum_2} = \boldsymbol{k})$ % $b = (k \geq c)$.
$\boldsymbol{N} \leftarrow \boldsymbol{N} \vee [\boldsymbol{Loop} \wedge \neg(\boldsymbol{N} + (\boldsymbol{N} \vee \boldsymbol{Loop}))]$
$\boldsymbol{N} \leftarrow \boldsymbol{N} \wedge \neg \boldsymbol{K}$
$\boldsymbol{K} \leftarrow \boldsymbol{K} \vee \boldsymbol{N}$
$\boldsymbol{Sum_1} \leftarrow \boldsymbol{Sum_1} + \neg(\uparrow_b \boldsymbol{K})$

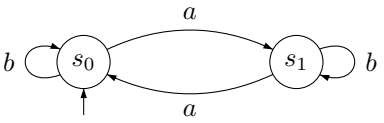
The values of the vector $\boldsymbol{\Delta h}$ can then be set to k using the mask $\boldsymbol{N}$. According to Theorem 1, when $k = c$, the initial state, the computation of $\boldsymbol{N}$ should use an alternative formula. But, in this case, the first bit of $\boldsymbol{N}$ is properly set by the first instruction since $\boldsymbol{Sum_1}$ contains the value c if $\Delta v_i = 0$.

Finally the algorithm computes the characteristic vector of state $2c$, the score, and the new value of $\boldsymbol{\Delta v}$.

5 Further Developments

As a first remark, we want to underline the fact that testing solvability for an automaton is a simple procedure that can be easily automated. If an automaton is solvable, it should also be possible to obtain automatically a corresponding vector algorithm, though not necessarily optimal. Indeed, in order to produce an efficient algorithm, Section 4 relied on the fact that the transition table of the automaton had many "arithmetic" symmetries. Is it possible to optimize the general algorithm?

Another interesting avenue is to broaden the class of automata that can be parallelized. Clearly, not all automata are solvable, the simplest counter-example being:



In this case, if one looks at an event x as a function from $x : Q \longrightarrow Q$, then both a and b are permutations. For an automaton to be solvable, it must have at least one *constant* event. One way to generalize the notion of decidability would be to extend it to constant *composition* of events, hinting at the possibility that properties of the syntactic monoid (see [9] and [7]) are related to the existence of vector algorithms.

References

1. A. Aho, *Algorithms for Finding Patterns in Strings*, in Handbook of Theoretical Computer Science, Vol. A, Elsevier (1990) 255-300.
2. R. A. Baeza-Yates and G. H. Gonnet, *A New Approach to Text Searching*, Communications of the ACM, 35, (1992) 74-82.
3. C.S. Calude, K. Salomaa, S. Yu, *Metric Lexical Analysis*, WIA99 Proceedings, Potsdam, July 1999, (to appear).
4. D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, (1997).
5. J. Holub and B. Melichar, *Implementation of Nondeterministic Finite Automata for Approximate Pattern Matching*, in Automata Implementation, LNCS 1660, Springer-Verlag, (1999).
6. E. Myers, *A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming* , J. ACM,46-3, (1999) 395-415.
7. J.-E. Pin, *Syntactic Semigroups*, in Handbook of Formal Languages, Vol. 1, Springer, (1997) 679-738.
8. P. H. Sellers, *The Theory and Computation of Evolutionary Distances*, J. of Algorithms, 1, (1980) 359-373.
9. H. Straubing, Finite Automata, Formal Logic and Circuit Complexity, Birkhauser, Boston, (1994).

Regularly Extended Two-Way Nondeterministic Tree Automata

Anne Brüggemann-Klein[1] and Derick Wood[2]

[1] Institut für Informatik, Technische Universität München, Arcisstr. 21, 80290 München, Germany. brueggem@informatik.tu-muenchen.de
[2] Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR. dwood@cs.ust.hk

Abstract. We establish that regularly extended two-way nondeterministic tree automata with unranked alphabets have the same expressive power as regularly extended nondeterministic tree automata with unranked alphabets. We obtain this result by establishing regularly extended versions of a congruence on trees and of a congruence on, so called, views. Our motivation for the study of these tree models is the Extensible Markup Language (XML), a metalanguage for defining document grammars. Such grammars have regular sets of right-hand sides for their productions and tree automata provide an alternative and useful modeling tool for them. In particular, we believe that they provide a useful computational model for what we call caterpillar expressions.

1 Introduction

We became interested in regularly extended two-way tree automata (tree automata that have a regular set of transitions instead of a finite set and, thus, unbounded degree nodes) because of our work [3] in which we show that tree languages recognized by caterpillar expressions are tree regular. Initially, we planned to prove this result by using regularly extended two-way tree automata to emulate caterpillar expressions and then applying the main theorem of this paper; namely, we generalize Moriya's result [8] that demonstrates finite two-way tree automata have the same expressive power as finite bottom-up tree automata to regularly extended tree automata. Our proof of this result is, however, very different from Moriya's. We first establish an algebraic characterization of the languages of regularly extended two-way tree automata and then show that the languages of regularly extended two-way tree automata satisfy the characterization. Unfortunately, we were unable to design a generic emulation of caterpillar expressions with regularly extended two-way tree automata. Therefore, we ended up using the algebraic characterization to to prove that caterpillar expressions recognize tree regular languages.

Regularly extended two-way tree automata are also of interest in their own right since they provide greater programming flexibility than do regularly extended one-way tree automata in much the same way that two-way finite-state automata do when compared to one-way finite-state automata. This choice is

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 57–66, 2001.

motivated by the Standard Generalized Markup Language (SGML) [7] and the Extensible Markup Language (XML) [2], which are metalanguages for document grammars that rely on these requirements. Although most work on classes of documents is grammatical in nature, grammars are not always the most appropriate tool for modeling applications. Murata [9] has argued that regularly extended tree automata often provide a more appropriate framework for investigating tree transformations, tree query languages, layout generation for trees, and context specification and evaluation.

The research on tree automata and regular languages of trees can be divided into two categories: one dealing with ranked and the other with unranked alphabets. The bulk of the literature deals with finite, ranked alphabets. Gécseg and Steinby [5] have written a comprehensive book on tree automata and tree transducers over ranked alphabets (an updated survey by the same authors appeared recently [6]); see also the text of Comon and his collaborators [4]. Although ranked and unranked alphabets are both finite, the transition relations of the corresponding tree automata for ranked alphabets can only be finite whereas the transition relations of the corresponding tree automata for unranked alphabets need not be finite. We consider the transition relation to be either regular or finite in the unranked case. We write **finite tree automaton** to mean that the tree automaton has a finite transition relation and we write **(regularly) extended tree automaton** to mean that the tree automaton has a regular transition relation.

Tree automata for unranked alphabets appear to have been first developed by Thatcher [12,13,14,15]. He states a number of results on finite tree automata that carry over directly from the theory of string automata. In particular, he developed the basic theory of finite tree automata and also introduced and investigated extended tree automata.

Other researchers studied various aspects of finite and extended tree automata; see the work of Barrero [1], Moriya [8], Murata [9] and Takahashi [11].

This paper has four further sections. In Section 2, we introduce the basic notation and terminology for extended tree automata, in Section 3, we introduce the notion of a top congruence and of views and, in Section 4, we use these notions to prove that extended two-way tree automata are only as expressive as extended bottom-up tree automata. Last, in Section 5, we state some conclusions and provide some research problems.

2 Notation and Definitions

We first recall tree and tree automata concepts before introducing the new concepts that we need.

Definition 1. Trees *have at least one node; their node labels are taken from a finite alphabet Σ. We represent trees by expressions that use the symbols in Σ as operators.*

Operators have no rank, so they can have any number of operands, including none. For example, the term $a(a(a()a())a(a()a()))$ represents a complete binary tree of height two, whose nodes all have the label a. Observe that external nodes or leaves correspond exactly to those subterms of the form $a()$.

We denote symbols in Σ with a, strings over Σ with w and sets of strings over Σ (we call them string languages) with L. The Greek letter λ denotes the empty string. We denote trees with t and sets of trees (we call them tree languages) with T. Subscripted and superscripted variables have the same types as their base names.

Definition 2. *We define the set* $nodes(t)$ *of* **nodes of a tree** t *as a set of strings of natural numbers. Its definition is by induction on* t*:*

For a tree $a(t_1 \cdots t_n)$, $n \geq 0$, *we define*

$$nodes(a(t_1 \cdots t_n) = \bigcup_{1 \leq i \leq n} i \cdot nodes(t_i) \cup \{\lambda\}.$$

The nodes of a tree viewed as terms correspond to subterms. We denote nodes of trees with ν.

Definition 3. *The* **root node** $root(t)$ *of a tree* t *is defined as* λ*. For each node* ν *of* t *we define the set* $children(\nu)$ *of* ν*'s* **children** *as the set of all nodes* $\nu \cdot i$ *in* $nodes(t)$*.*

Definition 4. *A node* ν *of a tree* t *is a* **leaf** *if and only if* $children(\nu) = \emptyset$*. The set of leaves of* t *is denoted by* $leaves(t)$*.*

Definition 5. *For each node* ν *of a tree* t*, we denote the label of* ν *in* Σ *by* $label(\nu)$*. More precisely, for a tree* $t = a(t_1 \cdots t_n)$, $n \geq 0$, *we define:*

1. *The label of the root node* λ *in* t *is* a*.*
2. *The label of the node* $i \cdot s$ *in* t *is the label of the node* s *in* t_i*.*

We are now in a position to define the class of tree automata that we investigate.

Definition 6. *A* **(regularly) extended two-way (nondeterministic) tree automaton** M *is specified by a triple* (Q, δ, F)*, where* Q *is a finite set of states,* $F \subseteq Q$ *is a set of final or accepting states, and* $\delta \subseteq \Sigma \times Q^* \times Q \times \{u, d, s\}$ *is a transition relation that satisfies the condition that, for all* a *in* Σ*,* q *in* Q *and* m *in* $\{u, d, s\}$*, the set* $\{w \in Q^* \mid (a, w, q, m) \in \delta\}$ *is a regular set of strings over the alphabet* Q*.*

If, for all a *in* Σ*,* q *in* Q *and* m *in* $\{u, d, s\}$*, the set* $\{w \in Q^* \mid (a, w, q, m) \in \delta\}$ *is a finite set of strings over the alphabet* Q*, then* M *is a* **finite two-way tree automaton.**

Finite two-way tree automata have been investigated by Moriya [8], whereas our results are on regularly extended two-way tree automata.

We define the computations of a two-way tree automaton on a tree by sequences of configurations. A configuration assigns a state of the automaton to each node in a cut of the tree.

Definition 7. *A* **cut** C **of a tree** t *is a subset of* $nodes(t)$ *such that, for each leaf node* ν *of* t*, there is exactly one node in* C *on the path from the root to* ν*; in other words, there is exactly one node in* C *given by a prefix of* ν*.*

Definition 8. *A* **configuration** c **of a two-way tree automaton** $M = (Q, \delta, F)$ *operating on a tree* t *is a map* $c : C \longrightarrow Q$ *from a cut* C *of* t *to the set of states* Q *of* M*.*

Let ν be a node of a tree t and let $c : C \longrightarrow Q$ be a configuration of the two-way tree automaton M operating on t. If $children(\nu) \subseteq C$, then formally $c(children(\nu))$ is a subset of Q. We overload this notation such that $c(children(\nu))$ also denotes the sequence of states in Q which arises from the order of ν's children in t.

Definition 9. *1. A* **starting configuration** *of a two-way tree automaton* $M = (Q, \delta, F)$ *operating on a tree* t *is a configuration* $c : leaves(t) \longrightarrow Q$ *such that* $c(\nu)$ *is any state* q *in* Q *such that* $(label(\nu), \lambda, c(\nu), u) \in \delta$*.*

2. A **halting configuration** *is a configuration* $c : C \longrightarrow Q$ *such that* $C = \{root(t)\}$*.*

3. An **accepting configuration** *is a configuration* $c : C \longrightarrow Q$ *such that* $C = \{root(t)\}$ *and* $c(root(t)) \in F$*.*

Definition 10. *1. A two-way tree automaton* $M = (Q, \delta, F)$ *operating on a tree* t *makes a* **transition** *from a configuration* $c_1 : C_1 \longrightarrow Q$ *to a configuration* $c_2 : C_2 \longrightarrow Q$ *(symbolically* $c_1 \longrightarrow c_2$*) if and only if it makes an up transition, a down transition or a no-move transition each of which we now define.*

2. M *makes an* **up transition** *from* c_1 *to* c_2 *if and only if* t *has a node* ν *such that the following four conditions hold:*

a) $children(\nu) \subseteq C_1$*.*

b) $C_2 = (C_1 \setminus children(\nu)) \cup \{\nu\}$*.*

c) $(label(\nu), c_1(children(\nu)), c_2(\nu), u) \in \delta$*.*

d) c_1 *is identical to* c_2 *on their domains' common subset* $C_1 \cap C_2$*.*

3. M *makes a* **down transition** *from* c_1 *to* c_2 *if and only if* t *has a node* ν *such that the following four conditions hold:*

a) $\nu \in C_1$*.*

b) $C_2 = (C_1 \setminus \{\nu\} \cup children(\nu))$*.*

c) $(label(\nu), c_2(children(\nu)), c_1(\nu), d) \in \delta$*.*

d) c_1 *is identical to* c_2 *on their domains' common subset* $C_1 \cap C_2$*.*

4. *M makes a* **no-move transition** *from c_1 to c_2 if and only if t has a node ν such that the following four conditions hold:*
   a) $\nu \in C_1$.
   b) $C_2 = C_1$.
   c) $(label(\nu), c_1(\nu), c_2(\nu), s) \in \delta$.
   d) *c_1 is identical to c_2 on $C_1 \setminus \{\nu\}$, which is equal to $C_2 \setminus \{\nu\}$.*

Definition 11. *1. A* **computation** *of a two-way tree automaton M on a tree t from configuration c to configuration c' is a sequence of configurations $c_1, \ldots, c_n$, $n \geq 1$, such that $c = c_1 \longrightarrow \cdots \longrightarrow c_n = c'$.*
2. An **accepting computation** *of M on t is a computation from a starting configuration to an accepting configuration.*

Definition 12. *1. A tree t is* **recognized** *by a two-way tree automaton M if and only if there is an accepting computation of M on t.*
2. The tree language $T(M)$ of a two-way tree automaton M is the set of trees that are recognized by M.

Definition 13. *A* **(regularly) extended (nondeterministic) bottom-up tree automaton** *is an extended (nondeterministic) two-way tree automaton $M = (Q, \delta, F)$ such that δ contains only transitions whose last component is u. For a bottom-up tree automaton M, we consider δ to be a subset of $\Sigma \times Q^* \times Q$ by dropping the fourth, constant component in the transition relation of a two-way tree automaton.*

Note that nondeterministic bottom-up tree automata are only as expressive as deterministic bottom-up tree automata [9].

Definition 14. *A tree language is* **regular** *if and only if it is the language of an extended bottom-up tree automaton.*

Clearly, since every extended bottom-up tree automaton is an extended two-way tree automaton, every regular tree language is recognized by some regular two-way tree automaton. Our goal is to prove that the converse also holds; namely, every tree language recognized by an extended two-way tree automaton is regular. We establish this result indirectly by developing an algebraic characterization of regular tree languages and then proving that the tree languages recognized by extended two-way tree automata satisfy this characterization.

3 Top Congruences

Definition 15. *A* **pointed tree** *(also called a tree with a handle or a handled tree) is a tree over an extended alphabet $\Sigma \cup \{X\}$ such that precisely one node is labeled with the variable X and that node is a leaf.*

Definition 16. *If t is a pointed tree and t' is a (pointed or nonpointed) tree, we can* **catenate** *t and t' by replacing the node labeled X in t with the root of t'. The result is the (pointed or nonpointed) tree tt'.*

Definition 17. *Let T be a tree language. Trees t_1 and t_2 are* **top congruent with respect to** *T ($t_1 \sim_T t_2$) if and only if, for each pointed tree t, the following condition holds:*

$$tt_1 \in T \text{ if and only if } tt_2 \in T.$$

The top congruence for trees is the tree analog of the left congruence for strings.

Lemma 1. *The top congruence is an equivalence relation on trees; it is a congruence with respect to catenations of pointed trees with nonpointed trees.*

Definition 18. *The* **top index of a tree language** *T is the number of $\sim_T$-equivalence classes.*

Lemma 2. *Each regular tree language has finite top index.*

A string language is regular if and only if it has finite index; however, that a tree language has finite top index is insufficient for it to be regular. For example, consider the tree language

$$L = \{a(b^i c^i) : i \geq 1\}.$$

Clearly, L has finite top index, but it is not regular. A second condition, regularity of local views, must also be satisfied.

Definition 19. *Let T be a tree language, a be a symbol in Σ, t be a pointed tree and T_f be a finite set of trees. Then, the* **local view of T with respect to t, a and T_f** *is the string language*

$$V_{t,a,T_f}(T) = \{t_1 \cdots t_n \in T_f^* \mid ta(t_1 \cdots t_n) \in T\}$$

over the alphabet T_f. For the purposes of local views we treat the trees in the finite set T_f as symbols in the alphabet T_f; the trees in T_f are primitive entities that can be catenated to give strings over T_f. Note that we are not catenating trees.

Lemma 3. *All local views of each regular tree language are regular string languages.*

Example 1. Let

$$T = \{c(t_1 \cdots t_n) \mid label(root(t_1)) \cdots label(root(t_n)) \in \{a^l b^l \mid l \geq 1\}\}.$$

The tree language T has top index four. Two of its equivalence classes are the sets of trees whose root labels are a or b; the other two are T and the set of trees that are not in T, but have the root label c. The local view of T with respect to the pointed tree $X()$, symbol c and the finite set of trees $\{a(), b()\}$ is the non-regular set of strings $\{a^l b^l \mid l \geq 1\}$. Hence, T has finite top index but it is not regular.

Theorem 1. *A tree language is regular if and only if it has finite top index and all its local views are regular string languages.*

At first glance it may appear that the local-view condition for regular tree languages is a condition on an infinite number of trees. But, if we exchange a tree t_1 in a finite set T_f by an equivalent—with respect to top congruence—tree t_2, then $V_{a,t,(T_f \setminus \{t_1\}) \cup \{t_2\}}(T)$ is the homomorphic image of $V_{a,t,T_f}(T)$ under a string isomorphism. Hence, if T has finite top index, we need to check the local-view condition for only a finite number of tree sets T_f.

4 Regularly Extended Two-Way Tree Automata Languages

Lemma 4. *The language of every extended two-way tree automaton has finite top index.*

Lemma 5. *The languages of all extended two-way tree automata have only regular local views.*

Proof. Let t be a pointed tree, a be a symbol in Σ, and T_f be a finite set of trees. We demonstrate that the local view $V_{t,a,T_f}(T)$ of T with respect to t, a, and T_f, namely the string language

$$\{t_1 \cdots t_n \in T_f^* \mid ta(t_1 \cdots t_n) \in T\},$$

is regular.

The proof is in three steps.

The first step is to recognize that V_{t,a,T_f} is a finite union of finite intersections of the following sets X_p and X_{pq}, $p, q \in Q$:

$$\begin{aligned} X_p = \{t_1 \cdots t_n \in T_f^* \mid & \\ & c_1 \longrightarrow c_2, \\ & c_1 \text{ is a starting configuration of } M \text{ on } a(t_1 \cdots t_n), \\ & c_2 \text{ is a halting configuration of } M \text{ on } a(t_1 \cdots t_n), \\ & c_2(root(a(t_1 \cdots t_n))) = p \text{ and} \\ & \text{there is no other halting configuration in the computation } c_1 \longrightarrow c_2\} \end{aligned}$$

and

$$X_{pq} = \{t_1 \cdots t_n \in T_f^* \mid$$
$$c_1 \longrightarrow c_2,$$
$$c_1 \text{ and } c_2 \text{ are halting configurations of } M \text{ on } a(t_1 \cdots t_n),$$
$$c_1(\mathit{root}(a(t_1 \cdots t_n))) = p,$$
$$c_2(\mathit{root}(a(t_1 \cdots t_n))) = q \text{ and}$$
$$\text{there is no other halting configuration in the computation } c_1 \longrightarrow c_2\}.$$

Any computation on $ta(t_1 \cdots t_n)$ from a starting configuration to a halting configuration can be partitioned into those parts that concern only t and those parts that concern only $a(t_1 \cdots t_n)$. The parts that concern only $a(t_1 \cdots t_n)$ form a computation from a starting configuration to a halting configuration, followed by a number of computations from halting configurations to halting configurations.

Hence, the $a(t_1 \cdots t_n)$-related parts of any accepting computation of M on $ta(t_1 \cdots t_n)$ first go from a starting configuration to a halting configuration, having M in some state p at $a(t_1 \cdots t_n)$'s root, and then from halting configuration to halting configuration, leading M from some state p_i to some state q_i on $a(t_1 \cdots t_n)$'s root until M finally leaves $a(t_1 \cdots t_n)$ and does not return. This implies that $t_1 \cdots t_n$ is in $X_p \cap X_{p_1 q_1} \cap \cdots \cap X_{p_r q_r}$. The state sequence $p, p_1, q_1, \ldots, p_r, q_r$ documents the behaviour of M at the root of $a(t_1 \cdots t_n)$ during an accepting computation of M on the complete tree $ta(t_1 \cdots t_n)$.

For any other sequence of trees $t'_1 \cdots t'_m$ in $X_p \cap X_{p_1 q_1} \cap \cdots \cap X_{p_r q_r}$, we can construct an accepting computation of M on $ta(t'_1 \cdots t'_m)$ by patching the $a(t_1 \cdots t_n)$-related parts of the original computation with computations on $a(t'_1 \cdots t'_m)$ that have the same state-behaviour at the root as $a(t_1 \cdots t_n)$ had. Since $t'_1 \cdots t'_m$ is in $X_p \cap X_{p_1 q_1} \cap \cdots \cap X_{p_r q_r}$, we can find such patches.

We conclude that the whole set $X_p \cap X_{p_1 q_1} \cap \cdots \cap X_{p_r q_r}$ is a subset of V_{t,a,T_f}.

Since there are only finitely many sets X_p and X_{pq}, the set V_{t,a,T_f} is a finite union of finite intersections of these.

The next two steps establish that X_p and X_{pq} are regular string languages.

First, a string $t_1 \cdots t_n$ is in X_p if and only if there are $p_1, \ldots, p_n$ in Q such that M, when operating on t_i beginning in a starting configuration, eventually reaches a halting configuration c such that $c(\mathit{root}(t_i)) = p_i$ and $(a, p_1 \cdots p_n, p, u) \in \delta$. The regularity of the transition table δ implies that X_p is a regular string language.

Second, let X^s_{pq} be the subset of X_{pq} in which the computation $c_1 \longrightarrow c_2$ (compare the definition of X_{pq}) makes just one computation step and let X^m_{pq} be the subset of X_{pq} in which the computation $c_1 \longrightarrow c_2$ makes more than one computation step. Then, X_{pq} is the (not necessarily disjoint) union of X^s_{pq} and X^m_{pq}. We demonstrate that both subsets are string regular.

First, note that

$$X^s_{pq} = \{t_1 \cdots t_n \in T_f^* \mid (a, p, q, s) \in \delta\}.$$

Hence, X^s_{pq} depends only on a and is either empty or T_f^*. In both cases, X^s_{pq} is regular.

Second, $t_1 \cdots t_n \in X_{pq}^m$ if and only if there are $p_1, \ldots, p_n, q_1, \ldots, q_n$ in Q such that $(a, p_1 \cdots p_n, p, d)$ is in δ and M, when operating on t_i, makes a computation from a halting configuration with root label p_i to a halting configuration with root label q_i and $(a, q_1 \cdots q_n, q, u) \in \delta$.

The regularity of the transition table δ implies that X_{pq}^m is a regular string language. □

Theorem 2. *The language of every extended two-way tree automaton is tree regular.*

5 Concluding Remarks

Moriya [8] uses crossing sequences to prove that finite two-way tree automata are as expressive as finite bottom-up tree automata. Thus, one follow-up question is whether we can prove our result using Moriya's approach.

We may define context-free two-way tree automata and ask whether they are as expressive as context-free bottom-up tree automata. Moriya [8] considers a pushdown variation on tree automata for which he demonstrates that the two-way version is indeed more expressive than the bottom-up version. Salomaa [10] proves that the yield languages of two-way pushdown tree automata are the recursively-enumerable languages.

Takahashi [11], on the other hand, establishes a different characterization of regular tree languages. We would be interested in knowing whether her characterization can be used to derive our algebraic characterization and, conversely, can we use our characterization to prove her's.

As we mention in the introduction, we originally planned to use extended two-way tree automata to emulate (or execute) caterpillar expressions. Our difficulty was that we could not design such an emulation. Therefore, is there an effective emulation of caterpillar expressions with extended two-way tree automata?

Acknowledgements. The authors would like to thank Kai Salomaa for bringing his own work on pushdown tree automata to our attention.

The work of both authors was supported partially by a joint DAAD-HK grant. In addition, the work of the second author was supported under a grant from the Research Grants Council of Hong Kong SAR.

References

1. A. Barrero. Unranked tree languages. *Pattern Recognition*, 24(1):9–18, 1991.
2. T. Bray, J. P. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. http://www.w3.org/TR/1998/REC-xml-19980210/, February 1998.
3. A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique, 2000. To appear in *Markup Languages*.

4. H. Comon, M. Daucher, R. Gilleron, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1998. Available on the Web from l3ux02.univ-lille3.fr in directory tata.
5. F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.
6. F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3, Beyond Words*, pages 1–68. Springer-Verlag, Berlin, Heidelberg, New York, 1997.
7. ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. International Organization for Standardization.
8. E. Moriya. On two-way tree automata. *Information Processing Letters*, 50:117–121, 1994.
9. M. Murata. Forest-regular languages and tree-regular languages. Unpublished manuscript, 1995.
10. K. Salomaa. Yield-languages of two-way pushdown tree automata. *Information Processing Letters*, 58:195–199, 1996.
11. M. Takahashi. Generalization of regular sets and their application to a study of context-free languages. *Information and Control*, 27(1):1–36, January 1975.
12. J. W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1:317–322, 1967.
13. J. W. Thatcher. A further generalization of finite automata. Technical Report RC 1846, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1967.
14. J. W. Thatcher. There's a lot more to finite automata theory than you would have thought. Technical Report RC 2852 (#13407), IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1970.
15. J. W. Thatcher and J. B. Wright. Abstract 65T-469. *Notices of the American Mathematical Society*, 12:820, 1965.

Glushkov Construction for Multiplicities

Pascal Caron[1] and Marianne Flouret[2]

[1] LIFAR, Université de Rouen, 76134 Mont-Saint-Aignan Cedex, France
Pascal.Caron@dir.univ-rouen.fr
[2] LIH, Université du Havre, 76058 Le Havre Cedex, France
Marianne.Flouret@univ-lehavre

Abstract. We present an extension to multiplicities of a classical algorithm for computing a boolean automaton from a regular expression. The Glushkov construction computes an automaton with $n + 1$ states from a regular expression with n occurences of letters. We show that the Glushkov algorithm still suits to the multiplicity case. Next, we give three equivalent extended step by step algorithms.

1 Introduction

First of all, let us recall which was the progression towards the results presented below and the underlying ideas. Several softwares have been developped concerning boolean automata. Let us cite AMoRE [11], Automate [4], Grail [15], ABOOL. For multiplicities, a Maple package called AMULT has been developped by Flouret and Laugerotte. Integration of the AMULT and ABOOL packages in the SEA environment [1] gathered the two underlying theories. Indeed, this environment allows the two packages to complete rational operations on automata, and also to go from the multiplicities theory to the boolean one and vice versa, by "suppressing" or "extending" coefficients. The classical rational operations are common for both theories (union and sum, concatenation and Cauchy product, Kleene's closure and star), although restrictions can be needed in some cases. We can then wonder whether similar algorithms could be applied on automata for both theories. Theoretical and algorithmic backgrounds enable us to compute automata from regular (rational) expression. Let us cite for the boolean case [2,9,13]. Several algorithms can compute an automaton from a regular expression. In the boolean case, a classical algorithm of interest is the Glushkov one [9] which leads us to compute an automaton with $n + 1$ states where n in the number of occurences of letters appearing in the expression. In the multiplicity case, the results of Schützenberger [16] giving the equivalence between rational and recognizable series, are the bases of construction algorithms demonstrated in [6]. This construction computes an automaton having a number of states of order $2n$ from a rational expression with n occurences of letters. Taking into account the efficiency of the Glushkov algorithm (in terms of size) we show that we can fit this algorithm in the case of multiplicities in any semiring. This new construction will be called the extended Glushkov construction. After some preliminaries, we extend in section 3 the Glushkov algorithm to the multiplicity

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 67–79, 2001.

case and, in section 4, the step by step construction. Our last results are to show the equivalence of these two constructions.

2 Theoretical Background

2.1 Definitions and Prerequisites

Let Σ be a finite alphabet, 1 the empty word, and $\mathbb{K}$ be a semiring. A formal series [3] is a mapping S from Σ^* into $\mathbb{K}$ usually denoted by $S = \sum_{w \in \Sigma^*} \langle S|w\rangle w$ (where $\langle S|w\rangle := S(w) \in \mathbb{K}$ is the coefficient of w in S). In the boolean case (resp. multiplicity case) simple rational operations are union (resp. sum), concatenation (resp. Cauchy product), and Kleene's closure (resp. star). An external product has to be defined in the case of multiplicities. Concatenation is extended to series by the convolution formula $R.S = \sum_{w \in \Sigma^*} \left(\sum_{uv=w} \langle R|u\rangle \langle S|v\rangle \right) w$. Remark that, if $\langle S|1\rangle = 0$, $S^* = \sum_{w \in \Sigma^*} \sum_{n \geq 0} \langle S^n|w\rangle w$ is well defined. We extend the rational closure to positive closure, S^+, taking $n > 0$.

A *regular language* (resp. *rational series*) is obtained from the letters by a finite number of combinations of the rational laws. The formula thus obtained is called a regular (resp. rational) expression of S.

A boolean automaton $\mathcal{M}$ over an alphabet Σ is usually defined [7,10] as a 5-tuple $(\Sigma, Q, I, F, \delta)$ where Q is a finite set of states, $I \subseteq Q$ the set of initial states, $F \subseteq Q$ the set of final states, and $\delta \subseteq Q \times \Sigma \times Q$ the set of edges. We denote by $L(E)$ the language represented by the regular expression E and by $L(\mathcal{M})$ the language recognized by the automaton $\mathcal{M}$. Kleene [12] asserts that it always exists an automaton $\mathcal{M}_E$ such that $L(E) = L(\mathcal{M}_E)$. This feature can be extended to the case of multiplicities in any semiring.

A $\mathbb{K}$-automaton [7] over an alphabet Σ is then a 5-tuple $(\Sigma, Q, I, F, \delta)$ on a semiring $\mathbb{K}$, and the sets I, F and δ are rather viewed as mappings $I : Q \to \mathbb{K}$, $F : Q \to \mathbb{K}$, and $\delta : Q \times \Sigma \times Q \to \mathbb{K}$. In fact, a $\mathbb{K}$-automaton is an automaton with input weights, output weights, and a weight associated to each edge. Here, for each word $w = a_1 \cdots a_p \in \Sigma^*$, the coefficient $\langle S|w\rangle$ is the sum of the weights of successful paths labeled by $a_1 \cdots a_p$, this weight being obtained by the product of input, output and edges weights of the path. This is equivalent (with $n = |Q|$) to the data of a triplet (λ, μ, γ) where $\lambda \in \mathbb{K}^{1 \times n}$ codes the input states, $\gamma \in \mathbb{K}^{n \times 1}$ codes the output states, and $\mu : \Sigma \to \mathbb{K}^{n \times n}$ codes the transition matrices for each letter $a \in \Sigma$ and is extended in a morphism from Σ^* to $\mathbb{K}^{n \times n}$, n being called the dimension of the representation. Then, a series S is *recognizable* if and only if it exists an automaton $\mathcal{M} = (\lambda, \mu, \gamma)$ such that its behavior $\sum_{w \in \Sigma^*} \lambda \mu(w) \gamma w$ is S. Schützenberger's classical theorem [16,8] asserts that rational series are exactly recognizable ones. This is then an extension of Kleene's classical result [12]. One can notice that we have to be very careful with the validity of rational

expressions. Indeed the star of a rational expression where the coefficient of the empty word is not 0 can not be a valid rational expression and then we can not compute in this case an automaton.

In this paper, we will use both notions of rational expression and series, as finite or infinite coding, according to the constructions.
There are several constructions of boolean automata from regular expression [9, 13,14,17]. In this paper we are interested in the Glushkov construction which allows us to have an automaton with $n+1$ states where n is the number of occurences of letters in the expression. The first step is to index each occurence of letter by its position in the expression E. The set of indices is named $Pos(E)$. Let $\overline{E}$ be the indexed expression. Glushkov defines four functions on $\overline{E}$ allowing us to compute a non necessarily deterministic automaton. $First(\overline{E})$ represents the set of initial positions of words of $L(\overline{E})$, $Last(\overline{E})$ the set of final positions of words of $L(\overline{E})$, $Follow(\overline{E}, i)$ the set of positions which immediately follows the position i in the expression $\overline{E}$. $Null(\overline{E})$ returns $\{\varepsilon\}$ if the language $L(\overline{E})$ recognizes the empty word, $\emptyset$ otherwise. Notice that, in the sequel of the paper we will write $First(E)$ for $First(\overline{E})$, as for the three other functions. These functions allow us to define the automaton $\overline{\mathcal{M}} = (\overline{\Sigma}, Q, s_I, F, \overline{\delta})$ where

1. $\overline{\Sigma}$ is the indexed alphabet,
2. $Q = Pos(E) \cup \{s_I\}$
3. $\forall i \in First(E),\ \overline{\delta}(s_I, a_i) = \{i\},\ \ a_i \in \overline{\Sigma}$
4. $\forall i \in Pos(E),\ \forall j \in Follow(E, i),\ \overline{\delta}(i, a_j) = \{j\},\ a_j \in \overline{\Sigma}$
5. $F = Last(E) \cup Null(E) \cdot s_I$

From $\overline{\mathcal{M}}$ we compute the automaton $\mathcal{M} = (\Sigma, Q, s_I, F, \delta)$ by replacing the indexed letters on edges by their corresponding letters in the expression E. For details on this construction see [18].

2.2 Classical $\mathbb{K}$ Constructions

In the following $\mathbb{K}$ is a commutative semiring.
Classical $\mathbb{K}$ matrix constructions have been given in [5] and proved in [6]. We just recall them for the usual rational operations.

Proposition 1. *Let R (resp. S) a rational series, $k \in \mathbb{K}$ and consider $(\lambda^r, \mu^r, \gamma^r)$ (resp. $(\lambda^s, \mu^s, \gamma^s)$) of dimension p (resp. q) an associated matrix representation. The linear representations of the external product, sum, concatenation and star are respectively*
$k \cdot R$:

$$\left(k \cdot \lambda^r, \left[\left(\mu^r(a)\right)_{a \in \Sigma}\right], \gamma^r\right), \tag{1}$$

$R + S$:

$$\left(\begin{pmatrix} \lambda^r & \lambda^s \end{pmatrix}, \left[\left(\begin{array}{c|c} \mu^r(a) & 0_{p \times q} \\ \hline 0_{q \times p} & \mu^s(a) \end{array}\right)_{a \in \Sigma}\right], \begin{pmatrix} \gamma^r \\ \gamma^s \end{pmatrix}\right), \tag{2}$$

R.S :

$$\left(\left(\lambda^r \; 0_{1\times q} \right), \left[\left(\begin{array}{c|c} \mu^r(a) & \gamma^r\lambda^s\mu^s(a) \\ \hline 0_{q\times p} & \mu^s(a) \end{array} \right)_{a\in\Sigma} \right], \begin{pmatrix} \gamma^r\lambda^s\gamma^s \\ \\ \gamma^s \end{pmatrix} \right) \quad , \tag{3}$$

If $\lambda^s\gamma^s = 0$, S^* :

$$\left(\left(0_{1\times q} \; 1 \right), \left[\left(\begin{array}{c|c} \mu^s(a) + \gamma^s\lambda^s\mu^s(a) & 0_{q\times 1} \\ \hline \lambda^s\mu^s(a) & 0 \end{array} \right)_{a\in\Sigma} \right], \begin{pmatrix} \gamma^s \\ 1 \end{pmatrix} \right) \quad . \tag{4}$$

We can notice that these constructions are always valid whatever the structure of the automata on which they are applied may be.

Proposition 2. *Let E be a rational expression in $\mathbb{K}$ such that $|E| = n$ is the number of its letters, $\mathcal{M}_E$ is the automaton obtained from E with the classical constructions, and $|\mathcal{M}_E|$ is the number of its states. Then $2n \leq |\mathcal{M}_E| \leq 3n$.*

The proof of this proposition is in the full version of the paper.

3 The Extended Glushkov Automaton

Let E be an expression over the alphabet Σ of a rational series with coefficients in $\mathbb{K}$. We index letters by their position in the expression. Let $\overline{E}$ be the new expression and $Pos(E) = \{i \in \mathbb{N}^* \mid a_i \in \overline{E}, a \in \Sigma\}$. For computing the automaton corresponding to a rational expression with the Glushkov algorithm, as in the classical (boolean) case [2,9], four recursive functions on $\overline{E}$ have to be defined.

3.1 Extended Definitions

The *Null* function allows us to know the coefficient of the empty word.

Definition 1. *The Null function can be defined recursively on the expression as follows: Let $k \in \mathbb{K}$.*

$$\begin{array}{ll} Null(\emptyset) & = 0 \\ Null(k) & = k \\ Null(a) & = 0 \\ Null(k\cdot F) & = k\cdot Null(F) \\ Null(F+G) & = Null(F) + Null(G) \\ Null(F\cdot G) & = Null(F)\cdot Null(G) \\ Null(F^+) & = 0 \\ Null(F^*) & = 1 \end{array}$$

We have to define an external product of a constant and a set of couples. Let $k \in \mathbb{K}$ and $X = \{(l_s, i_s) \mid l_s \in \mathbb{K}, i_s \in \mathbb{N}\}$. The product can then be written $k\cdot X = \{(k \times l_s, i_s)\}_{1\leq s\leq p}$ if $k \neq 0$, $0\cdot X = \emptyset$ and $X\cdot k = \{(l_s \times k, i_s)\}_{1\leq s\leq p}$ if $k \neq 0$, $X\cdot 0 = \emptyset$.

Then the states connected to the initial state, with their associated coefficient are given by the *First* function.

Definition 2. *First is recursively defined as follows:*

$$\begin{aligned}
First(\emptyset) &= \emptyset \\
First(k) &= \emptyset \\
First(a_i) &= \{(1,i)\} \\
First(k\cdot F) &= k\cdot First(F) \\
First(F+G) &= First(F)\cup First(G) \\
First(F\cdot G) &= First(F)\cup Null(F)\cdot First(G) \\
First(F^+) &= First(F) \\
First(F^*) &= First(F)
\end{aligned}$$

The terminal states with their associated output coefficients are given by the *Last* function.

Definition 3. *Last can be recusively defined as follows:*

$$\begin{aligned}
Last(\emptyset) &= \emptyset \\
Last(k) &= \emptyset \\
Last(a_i) &= \{(1,i)\} \\
Last(k\cdot F) &= Last(F) \\
Last(F+G) &= Last(F)\cup Last(G) \\
Last(F\cdot G) &= Last(F)\cdot Null(G)\cup Last(G) \\
Last(F^+) &= Last(F) \\
Last(F^*) &= Last(F)
\end{aligned}$$

For the following of the paper, we have to introduce the *Coeff* function which is defined on a set of couples. Let X be a set of couples (k,p), where $k\in\mathbb{K}$ and $p\in Q$, $Coeff_X(i)=\begin{cases} j \text{ if } (j,i)\in X \\ 0 \text{ elsewhere}\end{cases}$ and $Coeff_\emptyset(i)=0$. We will also use the $\mathcal{I}_X(i)$ function which is defined by $\mathcal{I}_X(i)=\begin{cases}1 \text{ if } i\in X \\ 0\ elsewhere\end{cases}$.

Now, the follow function allows us to compute the edges connecting the states each others as well as the corresponding multiplicities.

Definition 4. *Follow is recursively defined by: ($i\in\mathbb{N}^*$)*

$$\begin{aligned}
Follow(\emptyset,i) &= \emptyset \\
Follow(k,i) &= \emptyset \\
Follow(a_j,i) &= \emptyset \\
Follow(k\cdot F,i) &= Follow(F,i) \\
Follow(F+G,i) &= \mathcal{I}_{Pos(F)}(i)\cdot Follow(F,i)\cup\mathcal{I}_{Pos(G)}(i)\cdot Follow(G,i) \\
Follow(F\cdot G,i) &= \mathcal{I}_{Pos(F)}(i)\cdot Follow(F,i) \\
&\quad \cup\mathcal{I}_{Pos(G)}(i)\cdot Follow(G,i) \\
&\quad \cup Coeff_{Last(F)}(i)\cdot First(G) \\
Follow(F^+,i) &= Follow(F^*,i) \\
&= Follow(F,i)\cup Coeff_{Last(F)}(i)\cdot First(F)
\end{aligned}$$

All these functions allow us to define an extended Glushkov automaton.

Definition 5. *Let $\mathcal{M} = (\Sigma, Q, I, F, \delta)$ be the extended Glushkov automaton of the expression E. It is defined on a semiring $\mathbb{K}$ and on an alphabet Σ as follows:*

- $Q = Pos(E) \cup \{0\}$
- $I : Q \to \mathbb{K}$ *such that* $\begin{cases} 0 \to 1 \\ i \to 0 \end{cases}$
- $F : Q \to \mathbb{K}$ *such that* $\begin{cases} 0 \to Null(E) \\ i \to Coeff_{Last(E)}(i) \end{cases}$
- $\delta : Q \times \Sigma \times Q \to \mathbb{K}$
  $\delta(i, a, j) = Coeff_{Follow(E,i)}(j)$ *and* $a_j \in \overline{a}$ $(i \neq 0)$
  $\delta(0, a, j) = Coeff_{First(E)}(j)$ *and* $a_j \in \overline{a}$
  $\delta(i, a, 0) = 0$.

3.2 Matrix Extension of the Glushkov Construction

Definition 5 gives the computation of the extended Glushkov automaton. The linear representation (λ, μ, γ) is deduced from this definition. Its dimension is $|Q|$. For $0 \leq i, j \leq |Q| - 1$, $\lambda_i = I(i)$, $\gamma_i = F(i)$, and $\mu_{i,j}(a) = \delta(i, a, j)$.

Example:

Let $\mathbb{K} = \mathbb{Z}$. Let $E = 5[2ab + 3b \cdot 4(ab)^*]^*$ then we have $\overline{E} = 5[2a_1b_2 + 3b_3 \cdot 4(a_4b_5)^*]^*$. Let us compute the five functions defined above.

$$
\begin{aligned}
Null(E) &= Null(5 \cdot F^*) \\
&= 5 \cdot Null(F^*) \\
&= 5 \\
Last(E) &= Last(2ab + 3b \cdot 4(ab)^*) \\
&= Last(2ab) \cup Last(3b \cdot 4(ab)^*) \\
&= \{(1,2)\} \cup Last(b) \cdot Null(4(ab)^*) \cup Last(4(ab)^*) \\
&= \{(1,2),\ (4,3),\ (1,5)\} \\
First(E) &= 5 \cdot First(2ab + 3b \cdot 4(ab)^*) \\
&= 5 \cdot First(2ab) \cup 5First(3b \cdot 4(ab)^*) \\
&= 10 \cdot First(ab) \cup 15 \cdot First(b \cdot 4(ab)^*) \\
&= \{(10,1), (15,3)\} \\
Follow(E,1) &= Follow(2ab, 1) \\
&= Follow(a, 1) \cup \{(1,2)\} \\
&= \{(1,2)\} \\
Follow(E,2) &= Follow(2ab + 3b \cdot 4(ab)^*, 2) \cup First(2ab + 3b \cdot 4(ab)^*) \\
&= Follow(2ab + 3b \cdot 4(ab)^*, 2) \cup \{(2,1)\} \cup \{(3,3)\} \\
&= Follow(2ab, 2) \cup \{(2,1)\} \cup \{(3,3)\} \\
&= \{(2,1), (3,3)\} \\
Follow(E,3) &= Follow(2ab + 3b \cdot 4(ab)^*, 3) \cup \{(2,1), (3,3)\} \\
&= Follow(3b \cdot 4(ab)^*, 3) \cup \{(2,1), (3,3)\} \\
&= First(4(ab)^*) \cup \{(2,1), (3,3)\} \\
&= \{(4,4), (2,1), (3,3)\}
\end{aligned}
$$

$$
\begin{aligned}
Follow(E,4) &= Follow(2ab + 3b\cdot 4(ab)^*, 4) \\
&= Follow(3b\cdot 4(ab)^*, 4) \\
&= Follow((ab)^*, 4) \\
&= Follow(ab, 4) \\
&= \{(1,5)\} \\
Follow(E,5) &= Follow(2ab + 3b\cdot 4(ab)^*, 5) \cup \{(2,1),(3,3)\} \\
&= Follow((ab)^*, 4) \cup \{(2,1),(3,3)\} \\
&= \{(1,4),(2,1),(3,3)\}
\end{aligned}
$$

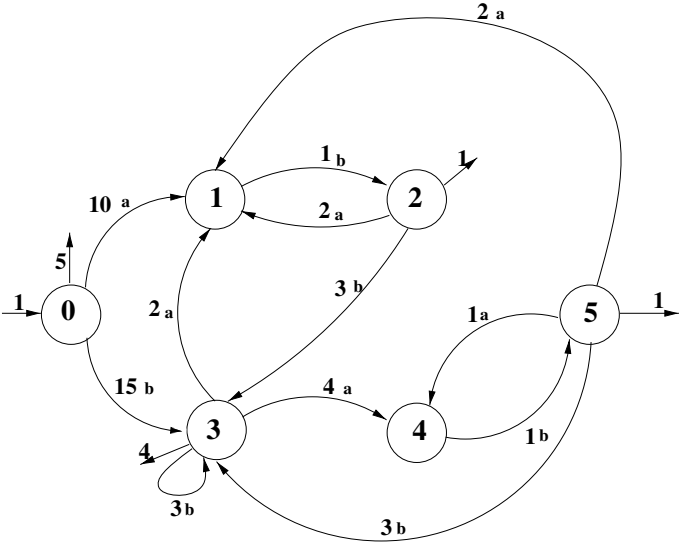


Fig. 1. Extended Glushkov automaton of the expression $E = 5[2ab + 3b\cdot 4(ab)^*]^*$

In terms of matrices representation, it can be written as follows

$$
\lambda = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \mu = \begin{pmatrix}
0 & 10 & 0 & 15 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 2 & 0 & 3 & 0 & 0 \\
0 & 2 & 0 & 3 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 2 & 0 & 3 & 1 & 0
\end{pmatrix}, \gamma = \begin{pmatrix} 5 \\ 0 \\ 1 \\ 4 \\ 0 \\ 1 \end{pmatrix}
$$

We can notice that for each letter a, the only non null columns are the ones indexed by j such that $a_j \in \overline{a}$. The matrix of the transition edges of the automaton can be seen as the superposition of the matrices for each letter, which justify the writing of μ above.

Proposition 3. *Every edge reaching a given state is labelled by the same letter.*

Proof. Extending the definition of δ to $\overline{\Sigma}$, the construction of the extended Glushkov automaton implies that if $l \neq j$, $\delta(i, a_l, j) = 0$. □

The construction of the extended Glushkov automaton leads to the following properties.

Proposition 4. *Let (λ, μ, γ) be the linear representation of an extended Glushkov automaton with $n+1$ states. This representation has the following properties:*

1. $\lambda = (1\ 0_{1\times n})$,
2. $\forall a \in \Sigma$, $\forall i \in [0..n]$, *such that* $\overline{a} = \{a_l \mid l \in [1..n]\}$ $\mu_{i,j}(a) = 0$ *if* $a_j \notin \overline{a}$.

Theorem 1. *The extended Glushkov automaton of an expression E recognizes the series denoted by E.*

Sketch of proof. This theorem is proved inductively. If $E = k$, the representation obtained from definition 5 can be written $\big((1)\ (0)\ (k)\big)$ and then $\lambda{\cdot}\,\mu(w){\cdot}\,\gamma = k$ if $w = \varepsilon$, 0 otherwise. We verify the same way for $E = a$ that $\lambda{\cdot}\,\mu(w){\cdot}\,\gamma = 1$ if $w = a$, 0 otherwise.Let F and G be two rational expressions representing the series S and T. We then suppose that the extended Glushkov automata $\mathcal{M}_F$ and $\mathcal{M}_G$ recognize respectively the series S and T. We prove that for $E = F + G$ the extended Glushkov automata $\mathcal{M}_{F+G}$ recognizes the series $S + T$. For $E = F.G$, we show that $\lambda_{S\cdot T}{\cdot}\,\mu_{S.T}(w){\cdot}\,\gamma_{S\cdot T} = \sum_{w=u\cdot v} (\lambda_S{\cdot}\,\mu_S(u){\cdot}\,\gamma_S) \times (\lambda_T{\cdot}\,\mu_T(v){\cdot}\,\gamma_T)$, and so on for $E = k{\cdot}\,F$ and $E = F^*$. □

4 Step by Step Construction

The step by step algorithm consists in computing a new automaton starting from automata with the following properties

- single initial state with no incoming edge,
- reduced automata.

This first property uses the proposition 4. We can then replace the mapping I by the state s_I which is the only one to have a non null input weight (which is 1).

We give here the three step by step algorithms corresponding to the rational operations Cauchy, Sum, Star. Let $\mathcal{M}_1 = (\Sigma_1, Q_1, s_1, F_1, \delta_1)$ and $\mathcal{M}_2 = (\Sigma_2, Q_2, s_2, F_2, \delta_2)$ be the automata of two series with the preceding propeties.

4.1 Sum Algorithm

The sum automaton $\mathcal{M}_3 = (\Sigma_3, Q_3, s_3, F_3, \delta_3) = \mathcal{M}_1 + \mathcal{M}_2$ is defined by the following algorithm[1].

[1] For all the algorithms, we write for simplicity $(p, a, q) \in \delta_i$ for $(p, a, q) \in Q_i \times \Sigma_i \times Q_i$

begin
$\Sigma_3 \leftarrow \Sigma_1 \cup \Sigma_2$
$Q_3 \leftarrow Q_1 \cup Q_2 \setminus \{s_2\}$
$s_3 \leftarrow s_1$
$F_3(s_3) \leftarrow F_1(s_1) + F_2(s_2)$
$F_3(q)_{q \in Q_1 \setminus \{s_1\}} \leftarrow F_1(q)$
$F_3(q)_{q \in Q_2 \setminus \{s_2\}} \leftarrow F_2(q)$
foreach $(p, a, q) \in \delta_1$ **do**
$\delta_3(p, a, q) \leftarrow \delta_1(p, a, q)$
end foreach
foreach $(p, a, q) \in \delta_2$ **such that** $p \neq s_2$ **do**
$\delta_3(p, a, q) \leftarrow \delta_2(p, a, q)$
end foreach
foreach $(s_2, a, q) \in \delta_2$ **do**
$\delta_3(s_3, a, q) \leftarrow \delta_2(s_2, a, q)$
end foreach
end

4.2 Cauchy Product Algorithm

$\mathcal{M}_3 = (\Sigma_3, Q_3, s_3, F_3, \delta_3) = \mathcal{M}_1 \cdot \mathcal{M}_2$ is computed as follows.

begin
$\Sigma_3 \leftarrow \Sigma_1 \cup \Sigma_2$
$Q_3 \leftarrow Q_1 \cup Q_2 \setminus \{s_2\}$
$s_3 \leftarrow s_1$
$F_3(q)_{q \in Q_2 \setminus \{s_2\}} \leftarrow F_2(q)$
if $F(s_2) \neq 0$ **then**
$F_3(q)_{q \in Q_1} \leftarrow F_1(q) \times F_2(s_2)$
else
$F_3(q)_{q \in Q_1} \leftarrow 0$
end if
foreach $(p, a, q) \in \delta_1$ **do**
$\delta_3(p, a, q) \leftarrow \delta_1(p, a, q)$
end foreach
foreach $(p, a, q) \in \delta_2$ **such that** $p \neq s_2$ **do**
$\delta_3(p, a, q) \leftarrow \delta_2(p, a, q)$
end foreach
foreach $p \in Q_1$ **such that** $F_1(p) \neq 0$ **do**
foreach $q \in Q_2$ **such that it exists** $a \in \Sigma_2$ **and** $(s_2, a, q) \in \delta_2$ **do**
$\delta_3(p, a, q) \leftarrow F_1(p) \times \delta_2(s_2, a, q)$
end foreach
end foreach
end

4.3 Star Algorithm

$\mathcal{M}_3 = (\Sigma_3, Q_3, s_3, F_3, \delta_3) = \mathcal{M}_1^*$.

begin
$\Sigma_3 \leftarrow \Sigma_1$
$Q_3 \leftarrow Q_1$
$s_3 \leftarrow s_1$
$F_3(s_3) \leftarrow 1$
$F_3(p)_{p \in Q_1 \setminus \{s_1\}} \leftarrow F_1(p)$
foreach $(p, a, q) \in \delta_1$ **do**
$\delta_3(p, a, q) \leftarrow \delta_1(p, a, q)$
end foreach
foreach $p \in Q_1$ **such that** $F_1(p) \neq 0$ **do**
foreach $q \in Q_1$ **such that it exists** $a \in \Sigma_1$ **and** $\delta_1(s_1, a, q) \neq 0$ **do**
$\delta_3(p, a, q) \leftarrow F_1(p) \times \delta_1(s_1, a, q)$
end foreach
end foreach
end

The proposition bellow gives the matrix constructions patterns for the rational operations. We will always consider the construction over an alphabet Σ. We need the following definitions. Let $Id_n^d = \begin{pmatrix} 0 & \cdots & 0 \\ & Id_n & \end{pmatrix}$, $Id_n^g = \begin{pmatrix} 0 & & \\ \vdots & Id_n & \\ 0 & & \end{pmatrix}$, and $L_n = \underbrace{\begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix}}_{n}$, with Id_n the square matrix identity of dimension n. For every matrix M of size $n+1$, $M \cdot Id_n^d$ deletes the first column, $Id_n^g \cdot M$ deletes the first row and $L_{n+1} \cdot M$ selects the first row.

Proposition 5. *Let R (resp. S) a rational series in $\mathbb{K}$ and let $\rho_r = (\lambda^r, \mu^r, \gamma^r)$ (resp. $\rho_s = (\lambda^s, \mu^s, \gamma^s)$) a corresponding linear representation of dimension $n+1$ (resp. $m+1$) with a single initial state with no incoming edge. The step by step algorithms lead to the following representations for $k \cdot R$, $R + S$, $R \cdot S$. For all cases, the resulting representation will be denoted (λ, μ, γ). The linear representations of the external product, the sum, the concatenation and the star are respectively*

$k \cdot R$:

$$\left(\begin{pmatrix} 1 & 0_{1\times n} \end{pmatrix} \left[\begin{pmatrix} 0 & kL_{n+1}\mu^r(a)Id_n^d \\ 0_{n\times 1} & Id_n^g \mu^r(a) Id_n^d \end{pmatrix}_{a\in\Sigma} \right], \begin{pmatrix} kL_{n+1}\gamma^r \\ Id_n^g \gamma^r \end{pmatrix} \right) ,$$

Denote $M = \begin{pmatrix} 1 & 0_{1\times n} & 0_{1\times m} \end{pmatrix}$, then we have:

$R + S$:

$$\left(M, \left[\left(\begin{array}{c|c|c} 0 & L_{n+1}\mu^r(a)Id_n^d & L_{m+1}\mu^s(a)Id_m^d \\ \hline 0_{n\times 1} & Id_n^g\mu^r(a)Id_n^d & 0_{n\times m} \\ \hline 0_{m\times 1} & 0_{m\times n} & Id_m^g\mu^s(a)Id_m^d \end{array} \right)_{a\in\Sigma} \right], \begin{pmatrix} L_{n+1}\gamma^r + L_{m+1}\gamma^s \\ Id_n^g\gamma^r \\ Id_m^g\gamma^s \end{pmatrix} \right) ,$$

$R\cdot S$:

$$\left(M, \left[\left(\begin{array}{c|c|c} 0 & L_{n+1}\mu^r(a)Id_n^d & L_{n+1}\gamma^r\lambda^s\mu^s(a)Id_m^d \\ \hline 0_{n\times 1} & Id_n^g\mu^r(a)Id_n^d & Id_m^g\gamma^r\lambda^s\mu^s(a)Id_m^d \\ \hline 0_{m\times 1} & 0_{m\times n} & Id_m^g\mu^s(a)Id_m^d \end{array}\right)_{a\in\Sigma}\right], \begin{pmatrix} L_{n+1}\gamma^r L_{m+1}\gamma^s \\ Id_n^g\gamma^r\lambda^s\gamma^s \\ Id_m^g\gamma^s \end{pmatrix}\right),$$

If $\lambda^s\gamma^s = 0$, S^* :

$$\left(\begin{pmatrix} 1 & 0_{1\times m} \end{pmatrix}, \left[\left(\begin{array}{c|c} 0 & L_{m+1}\mu^s(a)Id_m^d \\ \hline 0_{m\times 1} & Id_m^g\mu^s(a)Id_m^d + Id_m^g\gamma^s\lambda^s\mu^s(a)Id_m^d \end{array}\right)_{a\in\Sigma}\right], \begin{pmatrix} 1 \\ Id_m^g\gamma^s \end{pmatrix}\right) .$$

Sketch of proof. First, the elements of the linear representation associated to each operation are built by direct use of the algorithms. Next, we prove that these representations recognize the resulting series.

The complete proof is given in the full version of this paper. □

Corollary 1. *The step by step algorithms preserve the properties of single inital state and of trim automaton.*

The extended Glushkov construction can be written in terms of matrices representations. In fact, this way of computation consists in the substitution of operations on series (expressions) into operations over automata. The following proposition give the completely recursive representation of such automata, that is completely free from the calculus of the functions Null, First, Last and Follow and prove the equivalence of the two constructions.

Proposition 6. *Let* $(\lambda^k, \mu^k, \gamma^k) =$

$$\Big(\big(1\big), \Big[\big(0\big)_{a\in\Sigma}\Big], \big(k\big)\Big), \tag{5}$$

and $(\lambda^a, \mu^a, \gamma^a) =$

$$\left(\begin{pmatrix} 1 & 0 \end{pmatrix}, \left[\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}_{a\in\Sigma}, (0_{2\times 2})_{b\neq a\in\Sigma}\right], \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) \tag{6}$$

the representation of a.

With these basic constructions, the step by step algorithms build a Glushkov automaton from a rational expression.

Sketch of proof. Let F a rational expression with n occurences of letters, $(\lambda^f, \mu^f, \gamma^f)$ its extended Glushkov representation. For $F = k$ and $F = a$, representations (5) and (6) correspond by its definition to extended Glushkov representation, that is

$$\lambda^f = \begin{pmatrix} 1 & 0_{1\times n} \end{pmatrix}, \mu^f = \left[\begin{pmatrix} 0 & Coeff_{First(F)}(j)_{1\leq j\leq n} \\ 0_{n\times 1} & Coeff_{Follow(F,i)}(j)_{1\leq i,j\leq n} \end{pmatrix}_{a\in\Sigma}\right], \tag{7}$$

$$\gamma^f = \begin{pmatrix} Null(F) \\ Coeff_{Last(F)}(i)_{1 \leq i \leq n} \end{pmatrix}. \tag{8}$$

By induction on the length of rational expressions, we prove that their representations, built from (5) and (6) using patterns of proposition 5, are the Glushkov ones and then verify (7) and (8). □

5 Conclusion

In this paper, we have given a new algorithm for computing an automaton with multiplicities. We have verified that the boolean Glushkov construction suits to the multiplicity case with some adaptations. This new algorithm permits us to reduce from two to three times the size of the automaton.

References

1. P. Andary, P. Caron, J.-M. Champarnaud, G. Duchamp, M. Flouret, and E. Laugerotte. Sea: A symbolic environment for automata theory. In *Automata Implementation : Fourth International Workshop on Implementing Automata, WIA'99*, Lecture Notes in Computer Science, 1999. To be published.
2. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoret. Comput. Sci.*, 48(1):117–126, 1986.
3. J. Berstel and C. Reutenauer. *Rational series and their languages*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1988.
4. J.-M. Champarnaud and G. Hansel. Automate, a computing package for automata and finite semigroups. *J. Symbolic Comput.*, 12:197–220, 1991.
5. K. CulikII and J. Kari. Finite state transformations of images. In *Proceedings of ICALP 95*, volume 944. Lecture Notes in Computer Science, 1995.
6. G. Duchamp, M. Flouret, and É. Laugerotte. Operations over automata with multiplicities. In J.-M. Champarnaud and D. Ziadi, editors, *Automata Implementation: Third International Workshop on Implementing Automata, WIA'98*, volume 1660 of *Lecture Notes in Computer Science*, pages 183–191, 1999.
7. S. Eilenberg. *Automata, languages and machines*, volume A. Academic Press, New York, 1974.
8. M. Flouret. *Contribution à l'algorithmique non commutative*. PhD thesis, Université de Rouen, 1999.
9. V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
10. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.
11. V. Jansen, A. Potthoff, W. Thomas, and U. Wermuth. A short guide to the AMoRE system (computing Automata, MOnoids and Regular Expressions). Technical Report 90.2, Aachener Informatik-Berichte, Ahornstr 55, D–5100 Aachen, 1990.
12. S. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, Ann. Math. Studies 34:3–41, 1956. Princeton U. Press.
13. R. F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–57, March 1960.

14. B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.
15. D. Raymond and D. Wood. Grail, a C++ library for automata and expressions. *J. Symbolic Comput.*, 17:341–350, 1994.
16. M. P. Schützenberger. On the definition of a family of automata. *Inform. and Control*, 4:245–270, 1961.
17. K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11(6):419–422, 1968.
18. D. Ziadi, J.-L. Ponty, and J.-M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-déterministe. *Bull. Belg. Math. Soc.*, 4:177–203, 1997.

Implicit Structures to Implement NFA's from Regular Expressions

Jean-Marc Champarnaud

Université de Rouen, LIFAR
F-76821 Mont-Saint-Aignan Cedex, France
champarnaud@dir.univ-rouen.fr

Abstract. The aim of this paper is to compare three efficient representations of the position automaton of a regular expression: the Thompson ε-automaton, the $\mathcal{ZPC}$-structure and the $\mathcal{F}$-structure, an optimization of the $\mathcal{ZPC}$-structure. These representations are linear w.r.t. the size s of the expression, since their construction is in $O(s)$ space and time, as well as the computation of the set $\delta(X, a)$ of the targets of the transitions by a of any subset X of states. The comparison is based on the evaluation of the number of edges of the underlying graphs respectively created by the construction step or visited by the computation of a set $\delta(X, a)$.

1 Introduction

An efficient implementation of NFA's computation is based on two main features: the data structure to represent the NFA and the process to compute the set $\delta(X, a)$ of the targets of the transitions by a of an arbitrary subset X of states. In the general case, the $\delta(X, a)$ sets are independent a priori and the NFA is memorized by a table whose (q, a)-entry is the set $\delta(q, a)$. The complexity of a NFA is therefore generally measured by the size of its transition table. According to this convention, the position automaton [7,9] of a regular expression of alphabetic width n has the same $O(n^2)$ complexity as an arbitrary automaton with n states, whereas the common follow sets automaton [8] has an $O(nlog^2(n))$ complexity.

However, this classical definition does not take in consideration the specific properties which allow to provide a more efficient implementation of some NFA's families. It is the case when the NFA deduces from a regular expression: the syntactic structure of the expression induces dependencies among the $\delta(X, a)$ sets. Thanks to this property, a shared representation of the information can be designed, which leads both to save memory space and to speed up the process to compute the $\delta(X, a)$ sets. For example, the position automaton of an expression of alphabetic width n can be implemented with an $O(n)$ space and time complexity, leading to an $O(n)$ computation of any $\delta(X, a)$ set. In this paper we examine three representations of the position automaton yielding this complexity: the Thompson ε-automaton [13], the $\mathcal{ZPC}$-structure [15,10,12,4], and an optimization of the $\mathcal{ZPC}$-structure, the $\mathcal{F}$-structure. We explain the relationship between

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 80–93, 2001.

these structures and compare the number of elementary operations involved by their construction and by the computation of the $\delta(X, a)$ sets. Notice that the relationship between the position automaton and the Thompson ε-automaton has been studied in [6]; our approach is a more algorithmic one: in particular, the Thompson ε-automaton is compared to other linear representations rather than to the position automaton itself.

In order to deepen the comparison, some assumptions are made concerning input regular expressions. These hypothesis are presented in the following section. In the Section 3, we recall the definition of the position automaton and the complexity of the computations performed on its table. The Section 4 is devoted to the $\mathcal{ZPC}$-structure: we present an inductive definition of this representation, and give an accurate analysis of its complexity. The construction of the Thompson ε-automaton is recalled in the Section 5. The Section 6 provides a comparison between the $\mathcal{ZPC}$-structure of an expression and its Thompson ε-automaton, based on the fact that the state graph of the ε-automaton is deduced from the syntax tree of the expression. The $\mathcal{F}$-structure and its complexity are described in the Section 7. In order to provide a visual comparison of the various constructions, the figures have been gathered in the Annex A.

2 Hypothesis

The complexity of a regular expression is generally measured by its *size* s, i.e. the length of its prefixed form, or by its (alphabetic) *width* w, i.e. the number of occurrences of alphabet symbols. An arbitrary regular expression, such as the argument of a pattern matching command, may contain an arbitrary number of empty set, empty word and Kleene star operator occurrences. Its complexity should be measured by s, which may be arbitrarily greater than w. It is the reason why the complexity of the structures we study are firstly given w.r.t. s. In practical applications, it is profitable to preprocess the input expression in order to reduce the number of empty set, empty word and Kleene star operator occurrences. We define *reduced* expressions and show that they have linearily dependent size and width.

Definition 1. *A regular expression E is said to be reduced if it is such that:*
– Either E is the expression $\emptyset$ or E contains no occurrence of the empty set.
– Either E is the expression ε or the empty word only occurs in subexpressions $F + \varepsilon$ or $\varepsilon + F$, with $F \neq \varepsilon$ ('$+$' is denoted by '$+_\varepsilon$' in these expressions).
– Two consecutive operations in E cannot be both either an '$*$' or a '$+_\varepsilon$' operation.

Proposition 1. *Let E' be a regular expression of size s' and width w'. It is possible to construct a reduced regular expression E equivalent to E', of size s and width w, such that:*
- *The expression E is deduced from E' in $O(s')$ space and time.*
- *The expressions E and E' have an identical width: $w = w'$.*

- *If $E \neq \emptyset$ and $E \neq \varepsilon$, then E is such that:*
  - *The overall number of '$\cdot$' and '$+$' operators is equal to $w - 1$.*
  - *The overall number of '$*$' and '$+_\varepsilon$' operators is bounded by $2w - 1$.*
  - *The size of E is such that: $2w - 1 \leq s \leq 6w - 3$.*

The Thompson ε-automaton and the $\mathcal{ZPC}$-structure are known to be linear structures w.r.t. s. If the input expression is a reduced one, the complexity can be expressed w.r.t. w and thus made more precise. Moreover, a detailed implementation will be provided for each structure, in order to deduce an exact measure of the space. Lastly, the number of elementary operations will be bounded under the following assumption:
Hypothesis $\mathbf{H_1}$: The number of elementary operations to construct a structure (resp. to compute a $\delta(X, a)$ set) is proportional to the number of created (resp. visited) edges.

3 The Position Automaton of a Regular Expression

A regular expression is *linear* if and only if all its symbols are distinct. The following sets of symbols are associated to a linear expression E:
– $Null(E) = \{\varepsilon\}$ if $\varepsilon \in L(E)$ and $\emptyset$ otherwise,
– the set $First(E)$ of symbols matching the first symbol of some word in $L(E)$,
– the set $Last(E)$ of symbols matching the last symbol of some word in $L(E)$,
– the sets $Follow(E, x)$ of symbols following x in some word of $L(E)$, $\forall x \in \Sigma$.

Now, let E be a regular expression over Σ. If a is the j^{th} occurrence of a symbol in E, a_j is the *position* associated to a. The set of positions of E is denoted by $Pos(E)$. The linear expression deduced from E by substituting each symbol by its position is denoted by $\overline{E}$. Let h be the mapping from $Pos(E)$ to Σ such that $h(x)$ is the symbol related to the position x. The sets of positions associated to E are straightforwardly deduced from the sets of symbols associated to $\overline{E}$: $Null(E) = Null(\overline{E})$, $First(E) = First(\overline{E})$, $Last(E) = Last(\overline{E})$ and, $\forall x \in Pos(E)$, $Follow(E, x) = Follow(\overline{E}, x)$.

The position automaton $\mathcal{P}_E$ of E is deduced from the sets $Null(E)$, $First(E)$, $Last(E)$ and $Follow(E, x)$ as follows.

Definition 2. *The position automaton of E, $\mathcal{P}_E = (Q, \Sigma, \delta, I, F)$, is defined by:*
- $Q = Pos(E) \cup \{0\}$, *where 0 is not in $Pos(E)$,*
- $I = \{0\}$,
- $F = \begin{cases} Last(E) & \text{if } Null(E) = \emptyset \\ Last(E) \cup \{0\} & \text{otherwise} \end{cases}$
- $\delta(0, a) = \{x \in First(E) \mid h(x) = a\}$, $\forall a \in \Sigma$,
- $\delta(x, a) = \{y \mid y \in Follow(E, x) \text{ and } h(y) = a\}$, $\forall x \in Pos(E)$, $\forall a \in \Sigma$.

Remark 1. (a) An expression E such that $Null(E) = \{\varepsilon\}$ is said to be *nullable*. (b) For all position y, the label of each transition going into y is equal to $h(y)$: the position automaton is said to be *homogeneous*.

The sets $Null(E)$, $First(E)$, $Last(E)$ and $Follow(E,x)$ can be computed according to recursive formulas similar to the following ones, which hold for the case $E = F \cdot G$:

$$Null(F \cdot G) = Null(F) \cap Null(G)$$

$$First(F \cdot G) = \begin{cases} First(F) \cup First(G) & \text{if } Null(F) = \{\varepsilon\} \\ First(F) & \text{otherwise} \end{cases}$$

$$Last(F \cdot G) = \begin{cases} Last(F) \cup Last(G) & \text{if } Null(G) = \{\varepsilon\} \\ Last(G) & \text{otherwise} \end{cases}$$

$$Follow(F \cdot G, x) = \begin{cases} Follow(F,x) \cup First(G) & \text{if } x \in Last(F) \\ Follow(F,x) \cup Follow(G,x) & \text{otherwise} \end{cases}$$

The complexity of the position automaton is the following. The size of the table is bounded by $w(w+1)$. A direct computation of the $Follow(E,x)$ sets is in $O(w^2)$ time, hence an $O(w^3)$ construction of the table. The computation of $\delta(x,a)$ can be performed in $O(w)$ time, via three means: the conversion of the expression into its star normal form [2], the lazy transition evaluation defined in [5], or the elimination of the redundant follow links in the $\mathcal{ZPC}$-structure [15, 10]. It leads to an $O(w^2)$ construction of the table. Moreover, the computation of $\delta(X,a)$, where X is an arbitrary set of states, is performed in $O(w^2)$ time using the table.

4 The $\mathcal{ZPC}$-Structure of a Regular Expression

The $\mathcal{ZPC}$-structure is described in [15,10]. It is made of two copies of the syntax tree and of a collection of links connecting their nodes and implementing the computation of the $First$, $Last$ and $Follow$ sets of the subexpressions of E.

4.1 Definition of the $\mathcal{ZPC}$-Structure

Let $E_0 = \$ \cdot (E) \cdot \#$, where '$' and '#' are not in Σ, and let $Pos(E_0) = \{x_0, x_1, x_2, ..., x_w, x_{w+1}\}$, with $h(x_0) =$ '$' and $h(x_{w+1}) =$ '#', be the set of positions of E_0. The $\mathcal{ZPC}$-structure of E is defined by: $\mathcal{ZPC}_E = zpc(E_0)$; the inductive construction of $zpc(E)$ is illustrated by Fig. 1. Notice that the definition we give here is slightly different from the original one [15,10]. The links used to compute the $Last$ sets have been discarded, since our main goal is to design an efficient pattern matcher. The graphical presentation has been modified too, to facilitate the comparison with the Thompson ε-automaton.

Let us give some indications concerning this construction. The two copies of the syntax tree of E_0 associated to the structure $\mathcal{ZPC}_E$ are denoted by $Firsts(E)$ and $Lasts(E)$. If F is a subexpression of E_0 the corresponding node in $Firsts(E)$ (resp. $Lasts(E)$) is denoted by φ_F (resp. λ_F). As recalled below, the computation of $\delta(X,a)$ involves a bottom-up traversal of $Lasts(E)$ and a top-down one of $Firsts(E)$. Hence the following implementation of $\mathcal{ZPC}_E$, illustrated by Fig. 3:

• an array *lpositions* of size $w+1$, such that *lpositions*$[k]$, $k = 0$ *to* w, is a pointer to the leaf λ_{x_k} associated to the position x_k (notice it is not necessary to store a pointer to the leaf $\lambda_{x_{w+1}}$);

• for a node λ in $Lasts(E)$:
  – a pointer $lparent$ to the parent node; if λ is the root of a tree in the forest $Lasts(E)$ then $lparent(\lambda)$ is set to NULL,
  – a pointer $follow$, from the son of a node '$*$' (resp. the left son of a node '$\cdot$') in $Lasts(E)$ to its copy (resp. the right son of its copy) in $Firsts(E)$,
  – a boolean $notvisited$ initialized to $true$ before any $\delta(X, a)$ computation;

• an array $froots$ of size s, such that $froot[i]$ is a pointer to the root of a tree in the forest $Firsts(E)$;

• for a node φ of $Firsts(E)$:
  – the pointers $fleftson$ and $frightson$ deduced from the syntax tree,
  – the pointers $fbegin$, $fend$ and $fnext$ to compute $First(\varphi)$,
  – an integer $ftree$, which is the index of the tree the node φ belongs to;

• for each leaf associated to a position:
  – the character $symbol$ and the integer $frank$ associated to the position.

4.2 Complexity of the Construction of $\mathcal{ZPC}_E$

The forests $Firsts(E)$ and $Lasts(E)$ are generally drawn with distinct nodes. They can however be implemented with a shared set of $s+2$ nodes. Each node is equipped with six pointers: $fleftson$, $frightson$, $fbegin$, $fend$, $lparent$ and $follow$, and three data: the booleans $null$ and $notvisited$ and the index $ftree$. Moreover, each of the $w+1$ leaves of $Firsts(E)$ is equipped with two data: the pointer $fnext$ and the character $symbol$. Lastly, the size of the array $lpositions$ is equal to $w+1$, and the size of the array $froots$, which is less than the number of operators '$\cdot$', is bounded by s if E is an arbitrary expression, and by w if E is reduced. Hence the following proposition:

Proposition 2. *Let e be the space taken up by the structure $\mathcal{ZPC}_E$. If the expression is an arbitrary one, we have $e \approx 8s+3w$. If the expression is a reduced one, we have $e \approx 7s+4w$ and $18w < e < 46w$.*

As far as the time is concerned, the computation of the boolean $null$ must be taken in account. Conversely, some pointers are only computed for a subset of the set of nodes: the overall number of non-null $fleftson$ and $frightson$ pointers added with the number of trees in the forest $Firsts(E)$ is equal to $s-1$, and the pointers $follow$ are only generated by the '$\cdot$' and '$*$' nodes. According to the Proposition 1, the overall number of operators '$\cdot$' and '$*$', and thus of non-null $follow$ pointers, is bounded by $3w-2$. Lastly, the computation of each pointer is in $O(1)$ time. Under the hypothesis $\mathbf{H_1}$, we get the following proposition:

Proposition 3. *Let t be the time taken up by the construction of the structure $\mathcal{ZPC}_E$. If the expression is an arbitrary one, we have $t \approx 7s+3w$. If the expression is a reduced one, we have $t \approx 7s$ and $14w < t < 42w$.*

4.3 Computation of $\delta(X, a)$

The computation of the $\delta(X, a)$ sets on a $\mathcal{ZPC}$-structure is examined in [12, 4]. Since the position automaton is homogeneous, $\delta(X, a)$ can be deduced in $O(w)$ time from $\delta(X) = \bigcup_{a\in\Sigma} \delta(X, a)$, according to the formula: $\delta(X, a) = \{y \in \delta(X) \mid h(y) = a\}$. Moreover, the set $Y = \delta(X)$ can be computed by the following algorithm:

Algorithm deltaZPC:

• **Step 1:** Compute the set Λ of nodes λ in $Lasts(E)$ such that $Last(\lambda) \cap X \neq \emptyset$ and there exists a $follow$ link exiting from node λ.
• **Step 2:** Compute the set Φ of nodes φ in $Firsts(E)$ such that there exists a $follow$ link in Λ entering in φ. The set $Y = \delta(X)$ is such that $Y = \bigcup_{\varphi\in\Phi} First(\varphi)$.
• **Step 3:** Deduce a set Φ' from Φ so that the set Y is computed according to the formula: $Y = \biguplus_{\varphi\in\Phi'} First(\varphi)$.

Example 1. Let $E = ((a \cdot (a + b + \varepsilon))^* \cdot b)^*$ and consider the structure $\mathcal{ZPC}_E$ of Fig. 3. Let $X = \{a_1, b_4\}$. The following sets are computed:
$\Lambda = \{\lambda_5, \lambda_4, \lambda_3, \lambda_2, \lambda_1\}$, $\Phi = \{\varphi_6, \varphi_4, \varphi_{11}, \varphi_2, \varphi_{\#}\}$, $\Phi' = \{\varphi_6, \varphi_2, \varphi_{\#}\}$ and $Y = \{a_1, a_2, b_3, b_4, \#\}$.

Let us examine the complexity of of the Algorithm *deltaZPC*. Let L_X be the set of the roots of the trees in $Lasts(E)$ containing at least one node of Λ. For all λ in L_X, λ_X is defined as the partial subtree rooted in λ and made of all the paths going from λ to the leaves labelled by a position in X. Similarily, let F_X be the set of the roots of the trees in $Firsts(E)$ containing *at least two* nodes of Φ. For all φ in F_X, φ_X is defined as the partial subtree rooted in φ and obtained by deletion of the subtrees rooted in a node belonging to Φ. The number of edges in λ_X (resp. φ_X) is denoted by $|\lambda_X|$ (resp. $|\varphi_X|$). The size of the set Λ (resp. Φ) is bounded by the number f_X of $follow$ links.

Proposition 4. *Under the hypothesis* $\mathbf{H_1}$*, the computation time of the different steps of the Algorithm deltaZPC is the following:*
• *Step 1:* $t_1 = 2\sum_{\lambda\in L_X} |\lambda_X|$,
• *Step 2:* $t_2 = f_X$,
• *Step 3: Computation of* Φ'*:* $t'_3 = \sum_{\varphi\in F_X} |\varphi_X|$*; Computation of* Y*:* $t''_3 < w + 2f_X$.

Remark 2. The 2 coefficient in t_1 is due to the necessary initialization of the boolean *notvisited* before each $\delta(X, a)$ computation. In Step 3, if a $Firsts(E)$ tree exactly contains one node of Φ, this node is straightforwardly added to Φ'. The computation of $First(\varphi)$, for all φ in Φ', visits the two edges $fbegin(\varphi)$ and $fend(\varphi)$. The size of Φ' is bounded by the size of Φ. Moreover the computation of Y as a disjoint union implies to visit $|Y| \leq w + 1$ $fnext$ edges. Hence the bound on the time t_3.

5 The i-Automaton of a Regular Expression

In an ε-automaton, instantaneous transitions, i.e. transitions on the empty word ε, are authorized. Thompson has designed in [13] the construction of an ε-automaton recognizing the language of a given expression. We here present the definition of the *i-automaton* of an expression, a variant of the Thompson ε-automaton. The interest of the i-automaton is that its structure is closely related to the syntax tree of the expression, which makes the comparison to the $\mathcal{ZPC}$-structure more visual.

5.1 Definition of the i-Automaton

The i-automaton $(Q, \Sigma, i_E, t_E, \delta_i)$ of the expression E is defined inductively by the schemas of Fig. 2, where the unlabelled edges correspond to ε-transitions. For each subexpression F of E, the inductive construction produces the two states i_F and t_F. The set of states of the i-automaton is the union of the set of states i_F and of the set of states t_F.

5.2 Properties of the i-Automaton

Let us first notice that the Thompson ε-automaton can be viewed as an optimization of the i-automaton: in the case $E = F \cdot G$, Thompson merges the states i_E and i_F, as well as the states t_E and t_G. The properties we state for i-automata in the following are well-known properties of Thompson ε-automata. The relationship between the i-automaton and the position automaton is stated by the following proposition, where the ε-closure of the state q is denoted by $\varepsilon(q)$. The proof is by induction on the size of E.

Proposition 5. *Let $Pos(E) = \{x_1, x_2, ..., x_p\}$ be the set of positions of E and $(Q, Pos(E), i_E, t_E, \delta_i)$ be the i-automaton of E. Let I (resp. T) the set of states i_{x_k} (resp. t_{x_k}) generated by the expressions x_k. The following properties hold:*

- *$Null(E)$ is equal to $\{\varepsilon\}$ iff there exists an ε-path from i_E to t_E,*
- *$First(E) = \varepsilon(i_E) \cap I$, $Last(E) = \varepsilon^{-1}(t_E) \cap T$,*
- *$Follow(E, x_k) = \{x_\ell \mid i_{x_\ell} \in \varepsilon(t_{x_k})\}$, $\delta(x_k, a) = \{x_\ell \mid t_{x_\ell} \in \delta_i(\varepsilon(t_{x_k}) \cap I, a)\}$.*

The complexity of an i-automaton is as follows. An i-automaton can be represented by a table with $2s$ entries. Each state (except t_E) is the origin either of one symbol-transition, either of one or two ε-transitions. Moreover, the computation of each transition is in $O(1)$ time. Therefore, the space and time taken up by the construction of the table are equal to $4s$. More precisely, according to the Proposition 1, and under the hypothesis $\mathbf{H_1}$, it can be proved that the time is bounded by $15w$ when the expression is reduced. Moreover, due to the fact that the number of edges going out of a state is bounded by 2, the computation of the set $\delta(X, a)$ is performed in $O(s)$ time.

6 $\mathcal{ZPC}$-Structure vs. i-Automaton

We now use the fact that the i-automaton is deduced from the syntax tree of the expression to show similarities and differences with the $\mathcal{ZPC}$-structure.

6.1 The Structure of the i-Automaton

Let I_E and T_E be two copies of the syntax tree of E. Let X_I (resp. X_T) be the set of nodes of I_E (resp. T_E). Let $\mathcal{G}_E = (U, V, \Sigma \cup \{\varepsilon\})$ be a labelled digraph such that $U = X_I \cup X_T$ and V deduces from the set Γ_I of the edges $ileftson$ and $irightson$ of I_E, the set Γ'_T of the edges $tparent$ of T_E and the following sets:
– E_1 : the set of the edges $irightson$ in I_E with a node '$\cdot$' as origin,
– E_2 : the set of the edges $tparent$ in T_E associated to the $tleftson$ edge of a node '$\cdot$',
– E_3 : the set of the edges from the son of a node '$*$' in T_E to its copy in I_E,
– E_4 : the set of the edges from the left son of a node '$\cdot$' in T_E to the right son of its copy in I_E,
– E_5 : the set of the edges from a node '$*$' in I_E to its copy in T_E,
– E_6 : the set of the edges from a leaf (symbol, empty word) in I_E to its copy in T_E.

We have: $V = (\Gamma_I \setminus E_1) \cup (\Gamma'_T \setminus E_2) \cup (E_3 \cup E_4 \cup E_5 \cup E_6)$. All the edges are labelled by ε except the symbol-edges of E_6. The set V can be computed through a traversal of the syntax tree. The following proposition directly deduces from the inductive definition of the i-automaton.

Proposition 6. *The graph $\mathcal{G}_E$ is isomorphic to the state graph of the i-automaton of E.*

The comparison between $\mathcal{G}_E$ and $\mathcal{ZPC}_E$ yields the following similarities:
• Identical sets of nodes (as far as $\mathcal{ZPC}_E$ is defined on two distinct copies).
• Identical sets of edges connecting the second copy to the first one: the set $E_3 \cup E_4$ in $\mathcal{G}_E$ is equivalent to the set of $follow$ links in $\mathcal{ZPC}_E$.
• Closely related sets of syntactic edges:
– the sets of $ileftson$ edges and of $fleftson$ edges are equivalent,
– the set of $tparent$ edges is equivalent to the set of $lparent$ edges minus the edges (λ_F, λ_E) in the case $E = F \cdot G \wedge Null(G) = \{\varepsilon\}$,
– the set of $irightson$ edges is equivalent to the set of $frightson$ edges minus the edges (φ_E, φ_G) in the case $E = F \cdot G \wedge Null(F) = \{\varepsilon\}$.

On the other hand, the main differences are the following:
• *The handling of the positions*: in $\mathcal{G}_E$, each leaf of I_E associated to a position is connected to the corresponding leaf of T_E by an edge labelled by the associated symbol. In $\mathcal{ZPC}_E$, the array $lpositions$ provides the adresses of the leaves associated to the positions (which are shared by $Firsts(E)$ and $Lasts(E)$).
• *The processing of the nullable subexpressions:* in $\mathcal{G}_E$, a subexpression F is nullable iff there exists at least one ε-path from i_F to t_F. The existence of such

a path is related to the ε-edges connecting ε-leaves in I_E to the corresponding leaves in T_E, and to the edges of the set E_5 coming from the processing of the starred expressions. In $\mathcal{ZPC}_E$, the boolean $null(F)$ is computed for each subexpression F.

• *The computation of the set* $First(F)$, for F a non nullable subexpression of E: in $\mathcal{G}_E$, we have: $First(F) = \varepsilon(i_F) \cap Y$; the computation of $First(F)$ implies the exploration of the subtree of I_E rooted in i_F. In $\mathcal{ZPC}_E$, $First(F)$ directly deduces from the pointers $fbegin$ et $fend$ associated to F and from $fnext$ links.

6.2 Complexity of the Construction of $\mathcal{G}_E$ and $\mathcal{ZPC}_E$

The two constructions are in $O(s)$ space and time. The following table makes this result more precise, by giving the space taken by each structure, and the construction time under the hypothesis $\mathbf{H_1}$. The input is either an arbitrary expression, or a reduced one. Identical assumptions have been made to evaluate the complexity of the two constructions (cf. Propositions 2 and 3).

	arbitrary expression		*reduced expression*	
	$\mathcal{G}_E$	$\mathcal{ZPC}_E$	$\mathcal{G}_E$	$\mathcal{ZPC}_E$
space	$4s + 2w$	$8s + 3w$	$10w < e < 26w$	$18w < e < 46w$
time	$4s + 2w$	$7s + 3w$	$6w < t < 17w$	$14w < t < 42w$

The construction of $\mathcal{ZPC}_E$ is about two times more expensive than the construction of $\mathcal{G}_E$. The question is to know whether the additional information it computes allows a fairly large speedup of $\delta(X, a)$ sets computation or not.

6.3 Computation of $\delta(X, a)$

The comparison of the operations performed to compute the sets $\delta(X, a)$ in $\mathcal{G}_E$ and in $\mathcal{ZPC}_E$ is made possible by the following proposition:

Proposition 7. *Let ρ and ρ' be two nodes in the same forest of $\mathcal{ZPC}_E$ and let r and r' be the corresponding nodes in $\mathcal{G}_E$. The following property holds: there exists a path from ρ to ρ' iff there exists an ε-path from r to r'.*

Proof. Let us first verify that if there does not exist a link $tparent(t_G)$, $t_G \neq t_E$, then we have: there exists a link $lparent(\lambda_G) = \lambda_F$ iff there exists an ε-path from t_G to t_F. There does not exist a link $tparent(t_G)$, $t_G \neq t_E$, iff $F = G \cdot H$. In this case, there exists a link $lparent(\lambda_G)$ iff $Null(H) = \{\varepsilon\}$. On the other hand, there exists an ε-path from i_H to t_H iff $Null(H) = \{\varepsilon\}$. Moreover, there exists a $follow$ link from t_G to i_H. Finally, we get: there exists a link $lparent(\lambda_G)$ iff there exists an ε-path from t_G to t_F. It can be proved in a similar way that if there does not exist a link $irightson(t_F)$ then we have: there exists a link $frightson(\varphi_F) = \varphi_H$ iff there exists an ε-path from i_F to i_H. □

Example 2. The Figures 3 and 4 which respectively represent the $\mathcal{ZPC}$-structure and the $\mathcal{G}$-structure of the expression $E = ((a \cdot (a + b + \varepsilon))^* \cdot b)^*$ show that:
– The ε-path $(t_5, i_6, i_8, i_{10}, t_{10}, t_8, t_6, t_4)$ acts as the *lparent* link (λ_5, λ_4).
– The ε-path (i_2, i_3, t_3, i_{11}) acts as the $frightson$ link $(\varphi_2, \varphi_{11})$.
– There exists no ε-path from t_3 to t_2 and no *lparent* link from λ_3 to λ_2.
– There exists no ε-path from i_4 to i_6 and no $frightson$ link from φ_4 to φ_6.

Finally, the computation of $\delta(X, a)$ in $\mathcal{G}_E$ amounts to compute sets which are equivalent to Λ, Φ and Y sets, as in the Algorithm *deltaZPC*, and with the following complexity:
• Step 1: The number of visited edges is greater in T_E than in $Lasts(E)$ due to the ε-paths which act as *lparent* or $frightson$ links.
• Step 3: (a) Since the boolean *notvisited* is used in I_E, each edge must be visited a second time due to a necessary re-initialization step.
(b) If a tree in $Firsts(E)$ contains only one node φ of Φ, then $First(\varphi)$ is obtained directly from the pointers $fbegin(\varphi)$ and $fend(\varphi)$ and from the $fnext$ linking. On the opposite, the corresponding subtree in I_E is explored.
(c) If a tree in $Firsts(E)$ contains at least two nodes of Φ, the subsets of edges of this tree respectively visited in each structure are complementary.

This comparison can be summarized as follows:

Proposition 8. *The average computation time of $\delta(X, a)$ is smaller when performed on the structure $\mathcal{ZPC}_E$ than on the structure $\mathcal{G}_E$; the ratio between the average number of edges visited in $Firsts(E)$ and in I_E is equal to* 2/3.

7 The Forest-Structure of an Expression

From now on, we assume that E is a reduced expression and that f is the number of follow links. The Figure 5 illustrates the construction of $\mathcal{F}_E$, the forest-structure of $E = ((a \cdot (a + b + \varepsilon))^* \cdot b)^*$.

The structure $\mathcal{F}_E$ deduces from $\mathcal{ZPC}_E$ by the following optimizations:
• *The handling of empty word occurrences:* Since E is a reduced expression, we consider the unary operator '$\diamond$' such that: $Null(F^\diamond) = \{\varepsilon\}$, $First(F^\diamond) = First(F)$, $Last(F^\diamond) = Last(F)$ and, $\forall x \in Pos(F)$, $Follow(F^\diamond, x) = Follow(F, x)$, and $F^\diamond$ is substituted to each occurrence of $F + \varepsilon$ and $\varepsilon + F$. Since there are no longer leaves associated to the empty word, the pointers $fbegin$ and $fend$ are necessarily non-null, hence a faster computation. Moreover the size of a reduced expression is now bounded by $4w - 2$.
• *The compression of the forests $Firsts(E)$ and $Lasts(E)$:* The forest $Lasts(E)$ (resp. $Firsts(E)$) is compressed so that only the heads (resp. the tails) of $follow$ links are kept. The following properties hold:
– the compressed forests do not necessarily share the same set of nodes,
– the number of sons of a node may be more than 2,
– in each forest, the number of nodes is equal to f and the number of leaves is bounded by $w + 1$.

Proposition 9. *Let e (resp. t) be the space (resp. time) taken up by the construction of the $\mathcal{F}_E$ structure. We have: e and t are approximatively equal to $7f + 3w$ and $3w < e, t < 24w$.*

Proposition 9 holds under the same assumptions as for the analysis of the structures $\mathcal{G}_E$ and $\mathcal{ZPC}_E$. Moreover, the Algorithm *deltaZPC* can be readily made suitable for a $\mathcal{F}_E$ structure. Due to the reduction of the size of the forests, the computation of a $\delta(X, a)$ set is faster on $\mathcal{F}_E$ than on $\mathcal{ZPC}_E$.

8 Conclusion and Perspectives

The comparative analysis of the structures $\mathcal{G}_E$, $\mathcal{ZPC}_E$ and $\mathcal{F}_E$ has been conceived has a preliminary work to their programmation and to an experimental study of the performances. For the construction step, the main results are the following:

	reduced expression		
	$\mathcal{G}_E$	$\mathcal{ZPC}_E$	$\mathcal{F}_E$
space	$10w < e < 26w$	$18w < e < 46w$	$3w < e < 24w$
time	$6w < t < 17w$	$14w < t < 42w$	$3w < t < 24w$

Concerning the computation of the $\delta(X, a)$ sets, we have shown that $\mathcal{F}_E$ is naturally more efficient than $\mathcal{ZPC}_E$, which is more efficient in average than $\mathcal{G}_E$.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.
2. A. Brüggemann-Klein, Regular Expressions into Finite Automata. *Theoret. Comput. Sci.*, 120(1993), 197–213.
3. D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'Algorithmique.* Masson, Paris, 1992.
4. J.-M. Champarnaud , D. Ziadi and J.-L. Ponty, Determinization of Glushkov automata. In J.-M. Champarnaud *et al.*, eds, WIA'98, *Lecture Notes in Computer Science*, 1660(1999), 57–68, Springer.
5. C.-H. Chang and R. Paige. From Regular Expressions to DFAs using Compressed NFAs, in Apostolico. Crochemore. Galil. and Manber. editors. *Lecture Notes in Computer Science*, 644(1992), 88-108.
6. G. Giammarresi , J.-L. Ponty and D. Wood, The Glushkov and Thompson Machines Reexamined. *Submitted.*
7. V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
8. J. Hromkovič, S. Seibert, and T. Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. In R. Reischuk and M. Morvan, eds, STACS 97, *Lecture Notes in Computer Science*, 1200(1997), 55–66, Springer.
9. R. F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–57, March 1960.

10. J.-L. Ponty, D. Ziadi, and J.-M. Champarnaud. A new quadratic algorithm to convert a regular expression into an automaton. In D. Raymond *et al.*, eds, WIA'96, *Lecture Notes in Computer Science*, 1260(1997), 109–119, Springer.
11. J.-L. Ponty. Algorithmique et implémentation des automates. Thèse, Rouen, France, 1997.
12. J.-L. Ponty. An efficient null-free procedure for deciding regular language membership. *Theoret. Comput. Sci.*, WIA'97 Special Issue, D. Wood and S. Yu, editors, 231(2000).
13. K. Thompson, Regular Expression Search Algorithm, *Comm. Assoc. Comput. Mach.* 11(1968) 419–422.
14. S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume I, Word, Language, Grammar, pages 41–110. Springer, Berlin, 1997.
15. D. Ziadi, J.-L. Ponty and J.-M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-déterministe, Journées Montoises (1995), *Bull. Belg. Math. Soc.*, 4:177-203, 1997.

ANNEX A: Figures

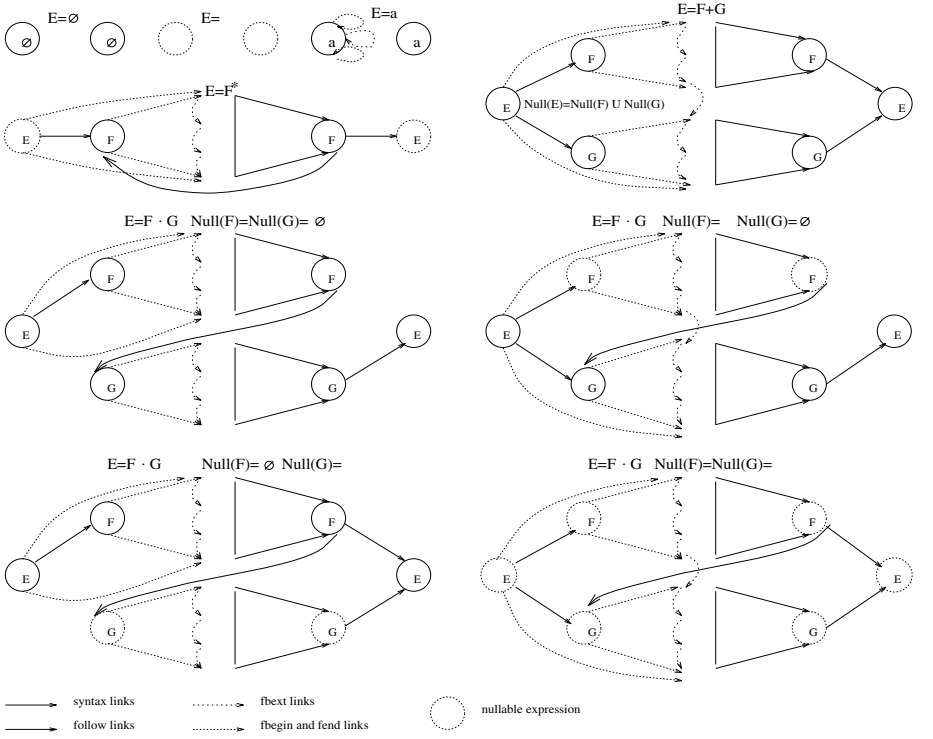


Fig. 1. The inductive definition of the structure $zpc(E)$.

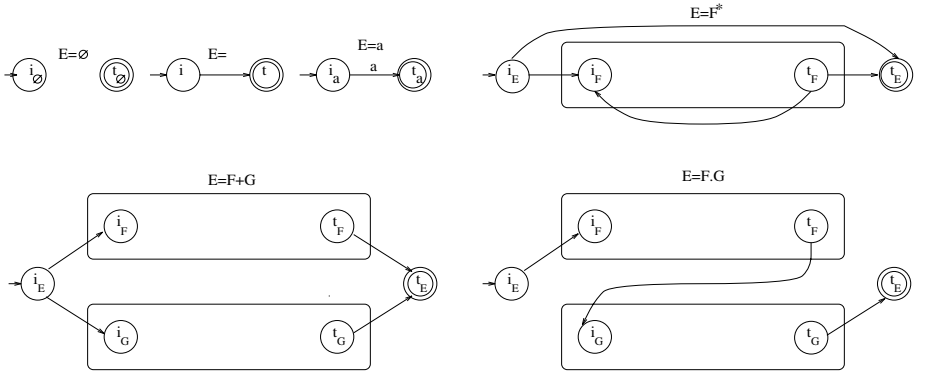


Fig. 2. The inductive definition of the i-automaton of a regular expression.

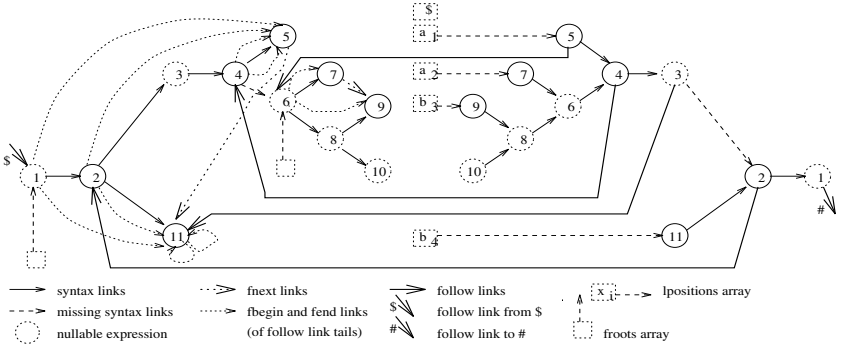


Fig. 3. The structure $\mathcal{ZPC}_E$ of $E = ((a \cdot (a + b + \varepsilon))^* \cdot b)^*$.

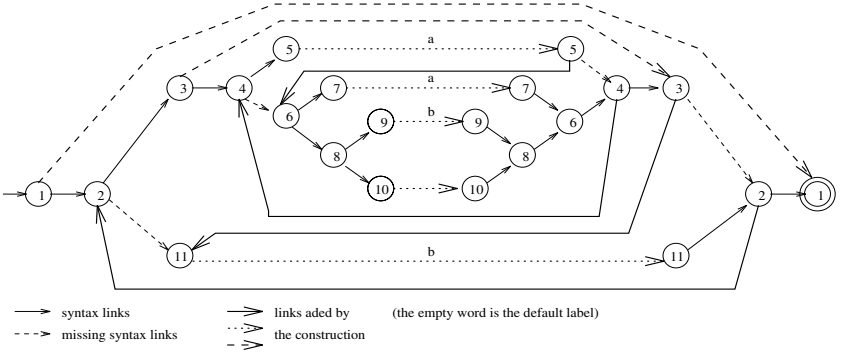


Fig. 4. The i-automaton of $E = ((a \cdot (a + b + \varepsilon))^* \cdot b)^*$.

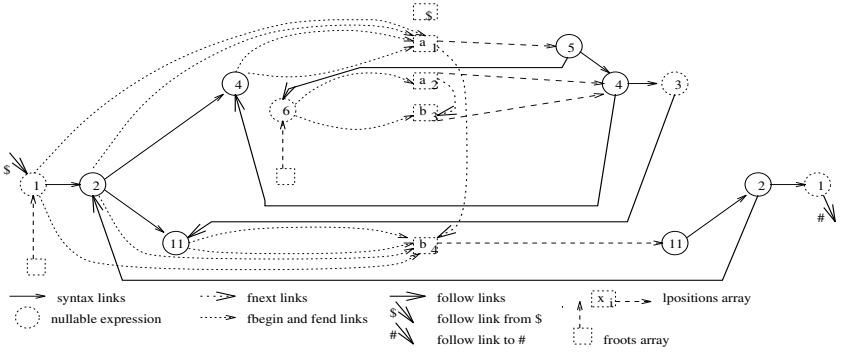


Fig. 5. The structure $\mathcal{F}_E$ of $E = ((a \cdot (a + b + \varepsilon))^* \cdot b)^*$.

New Finite Automaton Constructions Based on Canonical Derivatives

Jean-Marc Champarnaud and D. Ziadi

Université de Rouen, LIFAR
F-76821 Mont-Saint-Aignan Cedex, France
{champarnaud,ziadi}@dir.univ-rouen.fr

Abstract. Two classical constructions to convert a regular expression into a finite non-deterministic automaton provide complementary advantages: the notion of position of a symbol in an expression, introduced by Glushkov and McNaugthon-Yamada, leads to an efficient computation of the position automaton (there exist quadratic space and time implementations w.r.t. the size of the expression), whereas the notion of derivative of an expression w.r.t. a word, due to Brzozowski, and generalized by Antimirov, yields a small automaton. The number of states of this automaton, called the equation automaton, is less than or equal to the number of states of the position automaton, and in practice it is generally much smaller. So far, algorithms to build the equation automaton, such as Mirkin's or Antimirov's ones, have a high space and time complexity. The aim of this paper is to present new theoretical results allowing to compute the equation automaton in quadratic space and time, improving by a cubic factor Antimirov's construction. These results lay on the computation of a new kind of derivative, called canonical derivative, which makes it possible to connect the notion of continuation in a linear expression due to Berry and Sethi, and the notion of partial derivative of a regular expression due to Antimirov. A main interest of the notion of canonical derivative is that it leads to an efficient computation of the equation automaton via a specific reduction of the position automaton.

1 Introduction

Converting a regular expression into a finite automaton is a basic operation implemented inside many standard software tools such as compilers, context search requests, pattern matching engines, document processing packages and protocol simulators. Watson recently published a taxinomy on this topic [17], which aroused numerous theoretical and algorithmic developments [12,9,6,13, 16,5,8,4,2,19,11] since half a century.

Two notions turn out to be very suitable to carry the conversion of a regular expression into a non-deterministic automaton. The notion of position of a symbol in an expression, used in Glushkov [9] and McNaughton-Yamada [12] algorithms, leads to the computation of the classical position automaton of the

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 94–104, 2001.

expression. The notion of partial derivative of an expression w.r.t. a word, introduced by Antimirov [2], generalizes the notion of word derivative due to Brzozowski [6], and leads to the computation of the equation automaton of the expression.

For a given expression, with w occurrences of symbols, the number of states is equal to $w + 1$ in the position automaton, and is less than or equal to $w + 1$ in the equation automaton. Furthermore, in current applications, such as lexical analysis, the equation automaton can be much smaller than the position automaton. On the other hand, there exist several efficient implementations of the position automaton: the algorithms described in [4,8,19,15] have an $O(s^2)$ space and time complexity, where s is the size of the expression; on the opposite, there exist only two algorithms to compute the equation automaton, due to Mirkin [13] and to Antimirov [2], and they have a high space and time complexity: $O(s^3)$ space and $O(s^5)$ time for Antimirov's construction. The challenge is thus to design an algorithm to compute equation automata with the same efficiency as for computing position automata.

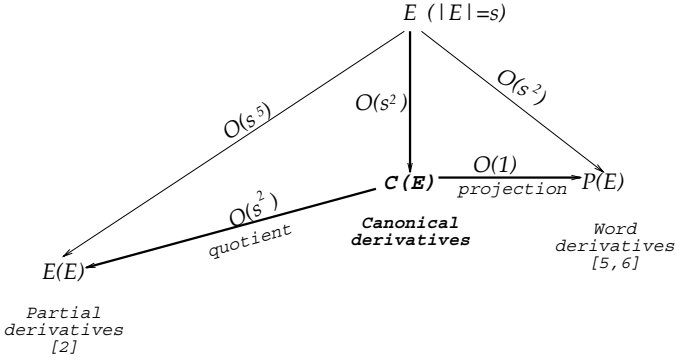
The notion of canonical derivative (or c-derivative) we introduce here allows one to study thoroughly the relation between the position automaton and the equation automaton and therefore leads to such an algorithm. The connection between the position automaton and the notion of word derivative has been studied by Berry and Sethi [5], who showed that a set of similar word derivatives is associated to each state of the position automaton (two similar derivatives deduce from each other by using associativity, commutativity and idempotence properties of the + operation). Canonical derivatives lead to a refinement of this property: we show that if two word derivatives of a linear expression (an expression where symbols are distinct) are similar, then the corresponding c-derivatives are identical, and we deduce that a unique expression, called a c-continuation, is associated to each state of the position automaton. Hence the definition of the c-continuation automaton, which contains the position automaton.

On the other hand, c-derivatives are connected to Antimirov's partial derivatives [2] in the following way: consider the set of c-continuations in the linearized version $\overline{E}$ of a regular expression E, and assume two c-continuations are equivalent if and only if they are linearizations of the same expression. Then the set of c-continuations in $\overline{E}$ is equal, modulo this equivalence, to the set of partial derivatives of E. Hence the computation of the equation automaton by reduction of the c-continuation automaton. Let us point out that these theoretical results are not related to the work of Hromkovič *et al.* [11]: firstly, the common follow sets automaton they define has more many states than the position automaton, and secondly they do not use any derivative-like tool; on the opposite, our results are based on a new algebraic tool: the c-derivatives.

These theoretical results have interesting algorithmic applications. It is shown in [7] that an implicit computation of the set of c-continuations and of the quotient set can be achieved with an $O(s^2)$ space and time complexity, which leads to a new quadratic implementation of both the positions and the equation automata and significantly improves the $O(s^5)$ complexity of Antimirov's

algorithm [2]. Notice that the techniques we use to handle c-continuations are necessarily different from the procedures used in [10] to implement the common follow sets automaton. On the other hand, some refinements to compute the set of transitions lay on the properties of the implicit structure [19,15] we designed in the past to represent the position automaton; a very closely related structure is used in [10].

Our investigations are illustrated by the following diagram. Our contribution is represented in bold font.



Section 2 recalls classical notions of automata theory. Section 3 summarizes theoretical results concerning c-derivatives and their relation with word derivatives. Relation with partial derivatives is examined in Sect. 4. The c-continuation automaton is introduced in Sect. 5, and the way it is connected to the position automaton and to the equation automaton is examined.

2 Preliminaries

We first recall terminology and basic results concerning regular languages and expressions, and finite automata. For further details about these topics, we refer to classical books [3,18].

Let Σ be a non-empty finite set of *symbols*, called the *alphabet*, and Σ^* be the set of all the *words* over Σ. Let ε be the *empty word*. A *language* over Σ is a subset of Σ^*. A *regular expression* over the alphabet Σ is 0, or 1, or a symbol $x_i \in \Sigma$, or is obtained by recursively applying the following rules: if F and G are two regular expressions, the *union* $(F + G)$ of F and G, the *product* $(F \cdot G)$ or FG of F and G, and the *star* (F^*) of F are regular expressions. The *regular language* $L(E)$ denoted by a regular expression E is such that: $L(0) = \emptyset$, $L(1) = \{\varepsilon\}$, $L(x_i) = \{x_i\}$ $\forall x_i \in \Sigma$, $L(F + G) = L(F) \cup L(G)$, $L(F \cdot G) = L(F)L(G)$ and $L(F^*) = L(F)^*$. Following [3], let $\mathcal{T}$ be the algebra of terms over the set $\Sigma \cup \{0, 1\}$, with the symbols of function $*, +, \cdot$, where $*$ is unary and $+$ and $\cdot$ are binary. Syntactical properties of the constants 0 and 1,

and of the operators $*$, $+$ and $\cdot$ lead to identification rules on the set of terms. In the following, we shall consider that the classical rules involving 0 and 1 hold: $0 + E = E = E + 0$, $1 \cdot E = E = E \cdot 1$, $0 \cdot E = 0 = E \cdot 0$. Associativity, commutativity and idempotence properties of the $+$ operation, called aci-rules, are captured by the notion of similarity [6]: two regular expressions F and G are aci-similar ($F \sim_{aci} G$) if and only if they reduce to the same expression by applying aci-rules.

We define $\lambda(E) = 1$ if $\varepsilon \in L(E)$, 0 otherwise. A regular expression E such that $\lambda(E) = 1$ is said to be *nullable*. We note $E \equiv F$ when the expressions E and F are identical. The size of E is noted $|E|$. The alphabetic width of E, i.e. the number of symbol occurrences in E, is noted $||E||$.

A regular expression E is *linear* over Σ if and only if every symbol of Σ occurs at most one time in E. If x is the j^{th} occurrence of symbol in E, the pair (x, j) is called a *position* of E, written x_j. The set of positions of E is denoted by $Pos(E)$. The expression $\overline{E}$ over $Pos(E)$ deduced from E by replacing symbol x in place j by x_j, for all j in $[1, \|E\|]$, is called *the linearized version* of E. $\overline{E}$ is linear over $Pos(E)$. We denote h the mapping from $Pos(E)$ to Σ such that $h(x_i) = x$, $\forall i \in [1, \|E\|]$, and $h(\overline{E}) \equiv E$. Let $E = a \cdot (a + b) + (a + b) \cdot (1 + b)$. We have: $Pos(E) = \{a_1, a_2, b_3, a_4, b_5, b_6\}$, $\overline{E} = a_1 \cdot (a_2 + b_3) + (a_4 + b_5) \cdot (1 + b_6)$, $h(a_1) = h(a_2) = h(a_4) = a$ and $h(b_3) = h(b_5) = h(b_6) = b$.

A *finite automaton* over Σ is a 5-tuple $\mathcal{M} = (Q, \Sigma, I, T, \delta)$ where Q is the set of *states*, I is the subset of *initial states*, T is the subset of *final states*, and δ is the *transition function*. $\mathcal{M}$ is *deterministic* ($\mathcal{M}$ is a DFA) if $|I| = 1$ and if δ maps $Q \times \Sigma$ to Q. Otherwise $\mathcal{M}$ is a *NFA* and δ maps $Q \times \Sigma$ to 2^Q.

3 Canonical Derivatives

Let E be a regular expression, a be a symbol in Σ and $u = u_1 \dots u_n$ be a word in Σ^*. The word derivative $u^{-1}E$ of E w.r.t. u is recursively defined as follows [6]:

$$
\begin{aligned}
a^{-1}0 = a^{-1}1 \quad &= \quad 0 \\
a^{-1}x = 1 \text{ if } a = x,\ 0 &\text{ otherwise} \\
a^{-1}(F + G) = a^{-1}F + a^{-1}G & \\
a^{-1}(F \cdot G) = a^{-1}F \cdot G \text{ if } \lambda(F) = 0,\ a^{-1}F \cdot G + a^{-1}G &\text{ otherwise} \\
a^{-1}(F^*) = a^{-1}F \cdot F^* & \\
\varepsilon^{-1}E = E \quad \textit{and} \quad (u_1 \dots u_n)^{-1}E \quad &= \quad (u_2 \dots u_n)^{-1}({u_1}^{-1}E)
\end{aligned}
$$

We introduce the notion of canonical derivative of a regular expression. The aim is to make it easy to compute derivatives of a linear expression. Let us recall that we assume that the classical rules involving involving 0 and 1 hold: $0 + E = E = E + 0$, $1 \cdot E = E = E \cdot 1$, $0 \cdot E = 0 = E \cdot 0$.

Definition 1 (c-derivative). *The c-derivative $d_u(E)$ of E w.r.t. u is recursively defined as follows:*

$$d_a(0) = d_a(1) \quad = \quad 0$$

$$
\begin{aligned}
d_a(x) &= 1 \text{ if } a = x,\ 0 \text{ otherwise} \\
d_a(F+G) &= d_a(F) \text{ if } d_a(F) \neq 0,\ d_a(G) \text{ otherwise} \\
d_a(F \cdot G) &= d_a(F) \cdot G \text{ if } d_a(F) \neq 0,\ \lambda(F) \cdot d_a(G) \text{ otherwise} \\
d_a(F^*) &= d_a(F) \cdot F^* \\
d_\varepsilon(E) &= E \quad \text{and} \quad d_{u_1 \ldots u_n}(E) \quad = \quad d_{u_2 \ldots u_n}(d_{u_1}(E))
\end{aligned}
$$

The following property is fundamental for handling c-derivatives of a linear expression. It deduces from Definition 1 by induction on the length of u and on the structure of E.

Proposition 1. *The c-derivative $d_u(E)$ of a linear expression E w.r.t. a word u of Σ^+ is either* 0 *or such that:*

$$d_u(u) = 1 \tag{1}$$

$$d_u(F+G) = d_u(F) \text{ if } d_u(F) \neq 0,\ d_u(G) \text{ otherwise} \tag{2}$$

$$d_u(F \cdot G) = \begin{cases} d_u(F) \cdot G & \text{if } d_u(F) \neq 0 \\ d_s(G) & \text{otherwise } (s \neq \varepsilon \text{ is some suffix of } u) \end{cases} \tag{3}$$

$$d_u(F^*) = d_s(F) \cdot F^* \quad (s \neq \varepsilon \text{ is some suffix of } u) \tag{4}$$

Proposition 2 states the properties of c-derivatives of a linear expression, and their relations to word derivatives.

Proposition 2. *Let E be a linear expression, a be any symbol in Σ, u and v be any word in Σ^*. The following properties hold:*
(a) A non-null c-derivative of E is either 1 *or a subexpression of E or a product of subexpressions.*
(b) $d_a(E) \equiv a^{-1}E$
(c) $d_u(E) \sim_{aci} u^{-1}E$
(d) $u^{-1}E \sim_{aci} v^{-1}E \Leftrightarrow d_u(E) \equiv d_v(E)$

Berry and Sethi [5] have proved that for any symbol a of a linear expression E, the non-null derivatives $(ua)^{-1}E$, for all u in Σ^*, are aci-similar. From Proposition 2 (d) we deduce the following theorem:

Theorem 1. *The set of non-null c-derivatives $d_{ua}(E)$ of a linear expression E reduces to a unique expression, the c-continuation of a in E, denoted $c_a(E)$.*

4 Canonical Derivatives and Partial Derivatives

Given a regular expression E and a symbol a, the set $\partial_a(E)$ of partial derivatives of E w.r.t. a is recursively defined on the structure of E as follows [2]:

$$
\begin{aligned}
\partial_a(0) = \partial_a(1) &= \emptyset \\
\partial_a(x) &= \{1\} \text{ if } a = x,\ \emptyset \text{ otherwise} \\
\partial_a(F+G) &= \partial_a(F) \cup \partial_a(G)
\end{aligned}
$$

$$\partial_a(F \cdot G) = \begin{cases} \emptyset & \text{if } G = 0 \\ \partial_a(F) \cdot G & \text{if } G \neq 0 \text{ and } \lambda(F) = 0 \\ \partial_a(F) \cdot G \cup \partial_a(G) & \text{otherwise} \end{cases}$$
$$\partial_a(F^*) = \partial_a(F) \cdot F^*$$

Any derivative of E can be represented by a finite set of some partial derivatives of E. For example, the two partial derivatives of $ab + a^*$ w.r.t. a are b and a^* and $a^{-1}(ab + a^*) = b + a^*$. Partial derivatives extend to words according to the formulas $\partial_\varepsilon(E) = \{E\}$ and $\partial_{ua}(E) = \partial_a(\partial_u(E))$.

We now examine the relation between c-derivatives and partial derivatives. Let E be a regular expression over Σ, $\overline{E}$ be the linearized version of E over $Pos(E)$ and h be the mapping it induces from $Pos(E)$ onto Σ. Let C_a be the set of the images by h of the non-null c-derivatives of $\overline{E}$ w.r.t. positions a_i such that $h(a_i) = a$. Proposition 3 says that C_a is the set of partial derivatives of E w.r.t. a, and Proposition 4 is used to extend this property to words.

Proposition 3. *Let E be a regular expression. Let H_E be a subexpression of E. Denote by $P(H_E)$ the property: $a_i \in Pos(E)(H_E) \wedge h(a_i) = a \wedge d_{a_i}(\overline{H_E}) \neq 0$. Then for all H_E one has:*

$$\bigcup_{P(H_E)} h(d_{a_i}(\overline{H_E})) = \partial_a(H_E)$$

Let E be a regular expression and $F = d_u(\overline{E})$ be a non-null c-derivative of $\overline{E}$. We consider the linearization $\overline{h(F)}$ of $h(F)$ and denote by h' the mapping from $Pos(h(F))$ onto Σ_E it induces. For example if $E = (ab + b)^*$ and $F = d_{a_1}(\overline{E}) = b_2(a_1b_2 + b_3)^*$, we have: $\overline{h(F)} = \overline{b(ab + b)^*} = b_1(a_2b_3 + b_4)^*$.
Since $F = H_1 \cdot H_2 \cdots H_l$, where H_i is a linear subexpression of $\overline{E}$, for $1 \leq i \leq l$, we have $\overline{h(F)} = H'_1 \cdot H'_2 \cdots H'_l$ and H_i and H'_i are two linearizations of the same expression. In our current example, we have: $H_2 = (a_1b_2 + b_3)^*$ and $H'_2 = (a_2b_3 + b_4)^*$.
Let a_j be the j^{th} symbol of $\overline{h(F)}$, and $\mu(a_j)$ be the j^{th} symbol of F. Notice that two distinct symbols of $\overline{h(F)}$ may be mapped to the same symbol of F: for instance, $\mu(b_1) = \mu(b_3) = b_2$. Proposition 4 shows that it is equivalent to compute the c-derivative of F w.r.t. a_i, and the c-derivative of $\overline{h(F)}$ w.r.t. any of the a_j such that $\mu(a_j) = a_i$.

We first state a lemma which is useful for the proof of this proposition. This lemma deduces from Proposition 1.

Lemma 1. *Let $d_u(\overline{E}) = H_1 \cdot H_2 \cdots H_l$ be a non-null c-derivative. For all i and j such that $1 \leq i < j \leq l$, if $Pos(E)(H_i) \cap Pos(E)(H_j) \neq \emptyset$, then there exists s a suffix of u such that $d_u(\overline{E}) = d_s(H_j) \cdot H_{j+1} \cdots H_l$.*

Proposition 4. *Let E be a regular expression, $F = d_u(\overline{E})$ be a non-null c-derivative of $\overline{E}$, and h' be the mapping associated to $\overline{h(F)}$. Let a_i be a position*

of E, and a_j be a position of $h(F)$ such that: $\mu(a_j) = a_i$, $h'(a_j) = h(a_i)$ and $d_{a_j}(\overline{h(F)}) \neq 0$. Then there exists m, $1 \leq m \leq l$, such that:

$$\begin{aligned} d_{a_j}(\overline{h(F)}) &= d_{a_j}(H'_m) \cdot H'_{m+1} \cdots H'_l \\ d_{a_i}(F) &= d_{a_i}(H_m) \cdot H_{m+1} \cdots H_l \\ h'(d_{a_j}(\overline{h(F)})) &= h(d_{a_i}(F)) \end{aligned}$$

Theorem 2. *Let E be a regular expression. Let H_E be a subexpression of E. $P(u, H_E)$ denotes the property: $v = \overline{u} \in Pos(E)^*(H_E) \wedge h(v) = u \wedge d_v(\overline{H_E}) \neq 0$. Then for all subexpression H_E of E, one has:*

$$\bigcup_{P(H_E,u)} h(d_{\overline{u}}(\overline{H_E})) = \partial_u(H_E)$$

Proof. The proof is by induction on the length of u. By Proposition 3 it is true for symbols. Assume now that the proposition is true for words u of length less than some integer n, $n > 1$, and prove it for words ua of length n. Denote by F the c-derivative $d_{\overline{u}}(\overline{H_E})$. By Proposition 4 we have:

$$\bigcup_{P(ua,H_E)} h(d_{\overline{u}a_i}(\overline{H_E})) = \bigcup_{P(u,H_E),\ P(a_j,h(F))} h'(d_{a_j}(\overline{h(F)}))$$

By Proposition 3, we get:

$$\bigcup_{P(u,H_E),\ P(a_j,h(F))} h'(d_{a_j}(\overline{h(F)})) = \partial_a(\bigcup_{P(u,H_E)} h(F)) \overset{\text{ind. hyp.}}{=} \partial_a(\partial_u(H_E))$$

□

5 Finite Automaton Constructions

This section enlightens the connections between three NFAs recognizing the language of a regular expression E: the position automaton $\mathcal{P}_E$, the equation automaton $\mathcal{E}_E$ and the c-continuation automaton $\mathcal{C}_E$. The construction of these automata is illustrated by Fig. 1.

5.1 The Position Automaton

Let E, $Pos(E)$, $\overline{E}$ and h be defined as usual. $L(E)$ is recognized by the automaton $\mathcal{P}_E$ [9,12], deduced from the positions sets $First$, $Last$ and $Follow$. $First(E)$ (resp. $Last(E)$) is the set of positions that match the first (resp. the last) symbol of some word in $L(\overline{E})$. $Follow(E, x)$, for all x in $Pos(E)$, is the set of positions that follow the position x in some word of $L(\overline{E})$.

Definition 2 (position automaton). *The position automaton of E, $\mathcal{P}_E = (Q, \Sigma, i, T, \delta)$, is defined by: $Q = Pos(E) \cup \{0\}$, $i = \{0\}$, $T = [$if $\lambda(E) = 0$ then $Last(E)$ else $Last(E) \cup \{0\}]$, $\delta(0, a) = \{x \in First(E) \mid h(x) = a\}$, $\forall a \in \Sigma$, and $\delta(x, a) = \{y \mid y \in Follow(E, x)$ and $h(y) = a\}$, $\forall x \in Pos(E)$ and $\forall a \in \Sigma$.*

5.2 The Equation Automaton

Antimirov proves in [2] that the cardinality of the set $\mathcal{PD}(E)$ of all partial derivatives of a regular expression E is less than or equal to $\|E\| + 1$. Hence the definition of $\mathcal{E}_E$, the equation automaton of E, whose states are the partial derivatives of E, and which recognizes $L(E)$.

Definition 3 (equation automaton). *The equation automaton of a regular expression E, $\mathcal{E}_E = (Q, \Sigma, i, T, \delta)$, is defined by: $Q = \mathcal{PD}(E)$, $i = E$, $T = \{p \mid \lambda(p) = 1\}$ and $\delta(p, a) = \partial_a(p)$, $\forall p \in Q$ and $\forall a \in \Sigma$.*

5.3 The c-Continuation Automaton

Let E, $\overline{E}$ and h be defined as usual. We assume 0 is a symbol not in $Pos(E)$. Let $c_0 = d_\varepsilon(\overline{E}) = \overline{E}$ and c_x be an abbreviation of $c_x(\overline{E})$. According to Theorem 1, the set of the c-continuations of the positions in E is finite. Thus we consider the non-deterministic automaton $\mathcal{C}_E$, called the c-continuation automaton of E, whose states are pairs (x, c_x) with x in $Pos(E) \cup \{0\}$, and whose transitions are deduced from the computation of c-continuations c_x.

Definition 4 (c-continuation automaton). *The c-continuation automaton of E, $\mathcal{C}_E = (Q, \Sigma, i, T, \delta)$, is defined by: $Q = \{(x, c_x) | x \in Pos(E) \cup \{0\}\}$, $i = (0, c_0)$, $T = \{(x, c_x) \mid \lambda(c_x) = 1\}$, $\delta((x, c_x), a) = \{(y, c_y) \mid h(y) = a$ and $d_y(c_x) \equiv c_y\}$, $\forall x \in Pos(E) \cup \{0\}$ and $\forall a \in \Sigma$.*

5.4 From $\mathcal{C}_E$ to $\mathcal{P}_E$ and $\mathcal{E}_E$

We now show that $\mathcal{C}_E$ and $\mathcal{P}_E$ are identical, as far as states of $\mathcal{C}_E$ are viewed as positions. More formally, let $p : Q \to Pos(E) \cup \{0\}$ such that $p(x, c_x) = x$. The automaton $p(\mathcal{C}_E)$ deduces from $\mathcal{C}_E$ by replacing Q by $Pos(E) \cup \{0\}$.

Theorem 3. *The automata $p(\mathcal{C}_E)$ and $\mathcal{P}_E$ of a regular expression E are identical.*

As a corollary, we deduce that $\mathcal{C}_E$ recognizes $L(E)$. The proof deduces from the following proposition:

Proposition 5. *Let E be a regular expression. The following equalities hold:*
(1) $First(E) = \{y \in Pos(E) \mid d_y(\overline{E}) \neq 0\}$;
(2) $Last(E) = \{y \in Pos(E) \mid \lambda(c_y(\overline{E})) = 1\}$;
(3) $Follow(E, x) = \{y \in Pos(E) \mid d_y(c_x(\overline{E})) \neq 0\}$.

We now explain how to deduce $\mathcal{E}_E$ from $\mathcal{C}_E$. Let $\sim$ be the equivalence relation on the set of states of $\mathcal{C}_E$, defined by: $(x, c_x) \sim (y, c_y) \Leftrightarrow h(c_x) \equiv h(c_y)$. Notice that two states (x, c_x) and (y, c_y) such that $c_x \neq c_y$ can be equivalent, as illustrated by the expression $E = de + fe$, which is such that $c_{d_1} = e_2 \neq e_4 = c_{f_3}$ and $h(c_{d_1}) = e = h(c_{f_3})$.

Let $[c_x]$ denote the class of the state (x, c_x). From Theorem 2, we deduce that the relation $\sim$ is right-invariant, i.e. for all a in Σ, for all (x, c_x), (t, c_t) in Q such that $(x, c_x) \sim (t, c_t)$, we have: $(\delta((x, c_x), a))/_\sim = (\delta((t, c_t), a)))/_\sim$. Hence the definition of the quotient automaton $\mathcal{C}_E/_\sim$.

Definition 5 (quotient automaton). *The automaton* $\mathcal{C}_E/_\sim = (Q_\sim, \Sigma, i, T, \delta)$ *is defined as follows:* $Q_\sim = \{[c_x] | x \in Pos(E) \cup \{0\}\}$, $i = [c_0]$, $T = \{[c_x] \mid \lambda(c_x) = 1\}$, *and* $[c_y] \in \delta([c_x], a) \Leftrightarrow \exists c_z \mid c_z \in [c_y]$, $h(z) = a$ *and* $d_z(c_x) \equiv c_z\}$, $\forall [c_x]$, $[c_z] \in Q_\sim$ *and* $\forall a \in \Sigma$.

The proof of the following theorem is based on Theorem 2.

Theorem 4. *Let E be a regular expression. The quotient automaton $\mathcal{C}_E/_\sim$ is isomorphic to $\mathcal{E}_E$, the equation automaton of E.*

Theorems 3 and 4 deepen the relations between the automata $\mathcal{P}_E$, $\mathcal{E}_E$, $\mathcal{C}_E$ and $\mathcal{C}_E/_\sim$.

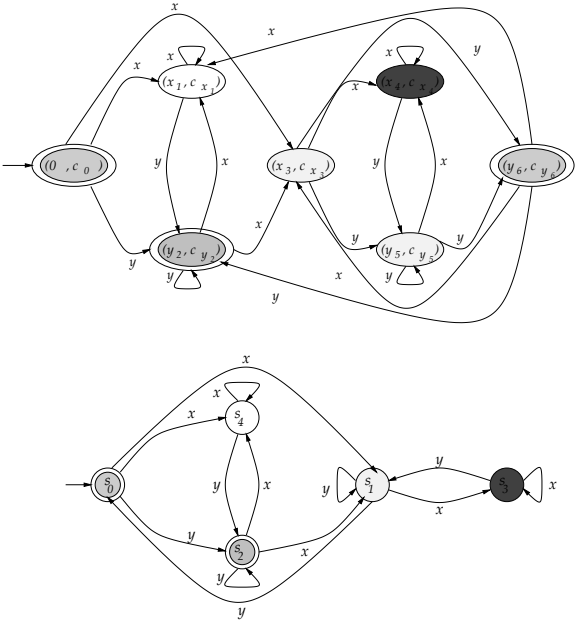


Fig. 1. $\mathcal{C}_E \simeq \mathcal{P}_E$ and $\mathcal{C}_E/_\sim \simeq \mathcal{E}_E$ for $E = ((x^*y)^* + x(x^*y)^*y)^*$.

Furthermore, they originate new algorithms for the construction of the position automaton and the equation automaton. These algorithms are described in [7].

6 Conclusion

The notion of canonical derivative of a regular expression allows to connect the notions of continuation in a linear expression and of partial derivative of a regular

expression; it leads to an efficient computation of sets of continuations and sets of partial derivatives. Thus it is a suitable tool for designing algorithms to convert a regular expression into a finite automaton: it can be used to compute both the position automaton and the equation automaton of a regular expression with an $O(s^2)$ space and time complexity. We are looking for relations over the set of c-continuations which would lead to smaller automata, with the same quadratic complexity.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
2. V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155:291–319, 1996.
3. D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'Algorithmique*. Masson, Paris, 1992.
4. A. Brüggemann-Klein, Regular Expressions into Finite Automata. *Theoret. Comput. Sci.*, 120(1993), 197–213.
5. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoret. Comput. Sci.*, 48(1):117–126, 1986.
6. J. A. Brzozowski. Derivatives of regular expressions. *J. Assoc. Comput. Mach.*, 11(4):481–494, 1964.
7. J.-M. Champarnaud and D. Ziadi, From C-Continuations to New Quadratic Algorithms for Automaton Synthesis. *Intern. Journ. of Alg. Comp.*, to appear.
8. C.-H. Chang and R. Paige. From Regular Expressions to DFAs using Compressed NFAs, in Apostolico. Crochemore. Galil. and Manber. editors. *Lecture Notes in Computer Science*, 644(1992), 88-108.
9. V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
10. Ch. Hagenah and A. Muscholl, Computing ε-free NFA from Regular Expressions in $O(nlog^2(n))$ Time, in: L. Prim *et al.* (eds.), MFCS'98, *Lecture Notes in Computer Science*, 1450(1998), 277-285, Springer.
11. J. Hromkovič, S. Seibert and T. Wilke, Translating regular expressions into small ε-free nondeterministic finite automata, in: R. Reischuk (ed.), STACS'97, *Lecture Notes in Computer Science*, 1200(1997), 55-66, Springer.
12. R. F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–57, 1960.
13. B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.
14. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), 1987.
15. J.-L. Ponty, D. Ziadi and J.-M. Champarnaud, A new Quadratic Algorithm to convert a Regular Expression into an Automaton, In: D. Raymond and D. Wood, eds., *Lecture Notes in Computer Science*, 1260(1997) 109–119.
16. K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11(6):419–422, 1968.
17. B. W. Watson. *Taxonomies and Toolkit of Regular Languages Algorithms*. PhD thesis, Faculty of mathematics and computing science, Eindhoven University of Technology, The Netherlands, 1995.

18. S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, vol. I, 41–110. Springer-Verlag, Berlin, 1997.
19. D. Ziadi, J.-L. Ponty, and J.-M. Champarnaud. Passage d'une expression rationnelle à un automate fini non-déterministe. Journées Montoises (1995), *Bull. Belg. Math. Soc.*, 4:177–203, 1997.

ANNEX A: Example

Let $E = ((x^*y)^* + x(x^*y)^*y)^*$. The automaton $\mathcal{E}_E$ of Fig. 1 is computed by the program *ProgCtoE* as follows.

```
E=((x*.y)*+x.(x*.y)*.y)*
size=16 alphabetic-width=6 #stars=5 alphabet=xy01+.*

----------Star subexpressions preprocessing (alphabet "xy01+.*")
Before identification : 5 star(s)
s1=x1*
s2=(x1*.y2)*
s3=x4*
s4=(x4*.y5)*
s5=((x1*.y2)*+x3.(x4*.y5)*.y6)*
After identification : 3 star(s)
S1={5} : ((x*.y)*+x.(x*.y)*.y)*
S2={2,4} : (x*.y)*
S3={1,3} : x*

----------Computation of the pseudo-continuations (alphabet "xy01+.*SSS")
Before identification : 7 pseudo-continuation(s)
l0=S1
l1=S3.y2.S2.S1
l2=S2.S1
l3=S2.y6.S1
l4=S3.y5.S2.y6.S1
l5=S2.y6.S1
l6=S1
After identification : 5 image(s) by mapping h
L0={0,6} : S1
L1={3,5} : S2.y.S1
L2={2} : S2.S1
L3={4} : S3.y.S2.y.S1
L4={1} : S3.y.S2.S1

----------The equation automaton
Initial state: 0       Final state(s): 0 2
Transitions:
State 0 :   x : 1 4    y : 2
State 1 :   x : 3      y : 0 1
State 2 :   x : 1 4    y : 2
State 3 :   x : 3      y : 1
State 4 :   x : 4      y : 2
```

Experiments with Automata Compression

Jan Daciuk

Alfa Informatica, Rijksuniversiteit Groningen
Oude Kijk in 't Jatstraat 26, Postbus 716
9700 AS Groningen, the Netherlands
j.daciuk@let.rug.nl

Abstract. Several compression methods of finite-state automata are presented and evaluated. Most compression methods used here are already described in the literature. However, their impact on the size of automata has not been described yet. We fill that gap, presenting results of experiments carried out on automata representing German, and Dutch morphological dictionaries.

1 Introduction

Finite-state automata are used in various applications. One of the reasons for this is that they provide very compact representations of sets of strings. However, the size of an automaton measured in bytes can vary considerably depending on the storage method in use. Most of them are described in [6], a primary reference for all interested in automata compression. However, [6] does not provide sufficient data on the influence of particular methods on the size of the resulting automaton. We investigate that in this paper. We used only deterministic, acyclic automata in our experiments. However, the methods we used do not depend on that feature. The automata we used were minimal (otherwise the first step in compression should be the minimization).

Our starting point is as follows. An automaton is stored as a sequence of transitions (fig. 1). The states are represented only implicitly. A transition has a label, a pointer to the target state, the number of transitions leaving the target state (*transition counter*), and a final marker. We use the transition counter to determine the boundaries of states, instead of finding them by subtracting addresses of states in a large vector of addresses of states as in [4], because we can get rid of the vector.

We store the final marker as the most significant bit of the transition counter. We use non-standard automata, *automata with final transitions* (see [2]), because they have less states and less transitions than the traditional ones. But since the storage methods are the same, the results are also valid for traditional automata.

2 Compression Techniques

Compression techniques fall into three main categories:

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 105–112, 2001.

	L	F	#	→
1	s		2	2
2	t		1	4
3	a		1	5
4	a		1	5
5	y	•	0	0

Fig. 1. Starting point storage method. L is a label, F marks final transitions, # is the number of outgoing transitions in the target state, and $\rightarrow$ is a pointer to the target state. The automaton recognizes words *say* and *stay*.

- coding of input data,
- making some parts of an automaton share the same space,
- reducing the size of some elements of an automaton.

Some techniques may contribute to the compression in two ways, e.g. changing the order of transitions in a state can both make sharing some transitions possible and reduce the size of some pointers. The first category depends on the kind of data that is stored in the automaton. The techniques we use apply to natural language dictionaries.

We define a deterministic finite-state automaton as $A = (\Sigma, Q, i, F, E)$, where Q is a finite set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $E \subseteq Q \times \Sigma \times Q$ is a set of transitions. We also define a function *bytes* that returns the number of bytes needed to store its argument: $bytes(x) = \lceil \log_{256} x \rceil$. Total savings (in percents) achieved by using a particular method M on the starting point automaton are $\eta^M(A) = 100\% \cdot \tau^M(A) \cdot \pi^M(A)/sizeof(A)$, where $\tau^M(A)$ is the number of transitions affected by the compression method, $\pi^M(A)$ is the saving in bytes per affected transition, and *sizeof(A)* is the size of the automaton in bytes. The size of an automaton in the starting point representation is $|E|(2 \cdot bytes|\Sigma| + bytes(|E|))$. In all those calculations, we assume that additional one-bit flags they require fit into the space taken by a pointer without the need to enlarge it.

2.1 Coding of Input Data

This section applies to natural language morphological dictionaries. Entries in such dictionaries usually contain 3 pieces of information: the inflected form, the base form, and the categories associated with the inflected form. It is common (e.g. in INTEX [11], [10], and in systems developed at the University of Campinas [5]) to represent that information as one string, with the base form coded. The standard coding consists of one character that says how many characters should be deleted from the end of inflected form so that the rest could match the beginning of the base form, and the string of characters that should be appended to the result of the previous operation to form the base form.

Such solution works very well for languages that do not use prefixes or infixes in their flectional morphology, e.g. French. However, in languages like German and Dutch, prefixes and infixes are present in many flectional forms. So to accommodate for this feature, we need 2 additional codes. The first one says what is the position from the beginning of a prefix or infix, the second code - the length of the prefix or infix. For languages that do not use infixes, but do use prefixes, it is possible to omit the position code.

2.2 Eliminating Transition Counters

There are two basic ways to eliminate transition counters. One uses a very clever sparse matrix representation (see [12], [8], and [9]). Apart from eliminating transition counters, it also gives shorter recognition times, as transitions no longer have to be checked one by one – they are accessed directly. However, that method excludes the use of other compression methods, so we will not discuss it here.

The other method (giving the same compression) is to see a state not as a set of transitions, but as a list of transitions ([6]). We no longer have to specify the transition count provided that each transition has a 1-bit flag indicating that it is the last transition belonging to a particular state (see fig. 2). That bit can be stored in the same space as the pointer to the target state, along with the final marker. We can combine that method with others.

$$\tau^{sb}(A) = |E|,\ \pi^{sb}(A) = bytes(|\Sigma|)$$

	L	F	S	→
1	s		•	2
2	t			4
3	a		•	5
4	a		•	5
5	y	•	•	0

Fig. 2. States seen as lists of transitions. S is the marker for the last transition in the state.

2.3 Transition Sharing

If we look at the figure 1, we can see that we have exactly the same transition twice in the automaton. However, once it is part of a state with 2 different transitions, and another time it is part of a state that has only 1 transition. As the information about state boundaries is not stored in the transitions belonging to the given state, we can share transitions between states (on the left on fig. 3). More precisely, a smaller state (with a smaller number of outgoing transitions) can be stored in a bigger one. It is also possible to place transitions of a state

so that part of them falls into one different state, and the rest into another one. This is possible only when we keep the transition counters, so we will not discuss that further.

	L	F	#	→
1	s		2	2
2	t		1	3
3	a		1	4
4	y	•	0	0

	L	F	S	→
1	s		•	2
2	t			3
3	a		•	4
4	y	•	•	0

Fig. 3. Two transitions (number 3 and 4 from figures 1 and 2) occupy the same space. Version with counters on the left, with lists – on the right.

In the version that uses lists of transitions, exactly one of the transitions belonging to a state holds information about one state boundary. The other boundary is defined by the pointer in transitions that lead to the state. If all transitions of a smaller state **A** are present as the last transitions of a bigger state **B**, then we can still store **A** inside **B** (on the right on fig. 3).

	L	F	N	#	→
1	s		•	2	
2	t			1	3
3	a		•	1	
4	y	•		0	0

	L	F	S	N	→
1	s		•	•	2
2	t				3
3	a		•	•	4
4	y	•	•		0

Fig. 4. The next flag with sharing of transitions. Version with counters on the left, with lists – on the right. *N* represents the next flag.

2.4 Next Pointers

Tomasz Kowaltowski *et al.* ([6]) note that most states have only one incoming and one outgoing transition, forming chains of states. It is natural to place such states one after another in the data structure. We call a state placed directly after the current one the *next state*. It has been observed in [6] that if we add a flag that is on when the target state is the next one, and off otherwise, then we do not need the pointer for transitions pointing to the next states. In case of the target being the next state, we still need a place for the flags and markers, but they take much less space (not more than one byte) than a full pointer. In case of the target state not being the next state, we use the full pointer. We need one additional bit in the pointer for the flag. Usually, we can find that space. In our implementation, we used the *next flag* only on the last transition of a state. Therefore, the representation of our example automaton looks like that given on figure 4.

The maximum number of transitions that can use next pointers is equal to the number of states in the automaton minus one, i.e. the initial state. The reason for this is that only one transition leading to a given state may be placed immediately in front of it in the automaton.

The transitions in states having more than one outgoing transition can be arranged in such a way that a transition leading to the next state in the automaton may not be the last one. However, if for a given transition its source state has exactly one outgoing transition, and its target state has exactly one incoming transition, the transition must use the next pointer.

Assuming $p, q, r \in Q$, and $a, b \in \Sigma$, we have:

$$|\{(p,a,q) : ((\forall_{(p,b,r)\in E} b = a, r = q) \wedge (\forall_{(r,b,q)\in E} b = a, r = s))\}| \leq \tau^{np}(A)$$

$$\tau^{np}(A) < |Q|, \quad \pi^{np}(A) = bytes(|E|) - 1$$

2.5 Tails of States

In section 2.3 we assumed that only entire states can share the same space as some larger states. When using the list representation (section 2.2), we can share only parts of states. We can have two or more states that share some but not all of their last transitions.

Let us consider a more complicated example (fig. 5). The transition number 4 holds in fact 3 identical transitions. The states reachable from the start state have both 2 transitions. One of those transitions is common to both states (the one with label a). The second one is different (either labeled with l or t). To avoid confusion, we did not use the *next* flag.

	L	F	S	→	↪
1	p			3	
2	s		•	5	
3	l			4	
4	a		•	6	
5	t			4	4
6	y	•	•	0	

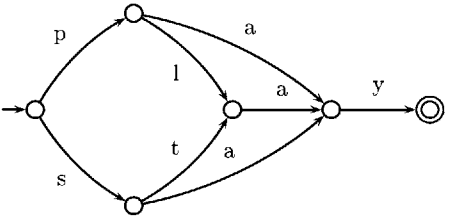


Fig. 5. Automaton recognizing words *pay*, *play*, *say*, and *stay*, and sharing last transitions of states. ↪ is a pointer to the tail of the state.

To implement tail sharing we need two things: a new flag (we call it the *tail flag*, not shown on the figure 5 as its value is implied), and an additional pointer occurring only when the tail flag is set. When the flag is set, then only the first transitions are kept, and the additional pointer points to the first transition of the tail shared with some other state. We need 1 bit for the flag, and we allocate a place for it in the bytes of transition pointers.

2.6 Changing the Order of Transitions

In the examples we showed so far, nothing was told about the order of transitions in a state. Techniques of transition sharing depend on the order of transitions. Automata construction algorithms (e.g. [3]) may impose initial ordering, but it may not be the best one. Kowaltowski *et al.* ([6]) propose sorting the transitions on increasing frequency of their labels. They also propose to change the order in each state individually. The order of transitions also influences the number of states that are to be considered as *next*. To increase the number of *next pointers*, we try to change the order of transitions in states that do not already have that compression. To increase transition sharing, we put every possible set of n transitions (starting from the biggest n) of every state into a register, and then look for states and tails of states that match them.

2.7 Other Techniques

There are other techniques that we have not experimented with. They include local pointers and indirect pointers. Local pointers are only mentioned in [8]. We can only stipulate that what they propose is 1-byte pointers for states that are located close in the automaton, and full-length pointers for other states. A flag is needed to differentiate among the two. Indirect pointers are proposed in US patent 5,551,026 granted August 27, 1996 to Xerox. By putting pointers to frequently referenced locations into a vector of full-length pointers, and replacing those pointers in transitions with (short) indexes in the vector, one can gain more space.

3 Experiments

3.1 Data

Our experiments were carried out on morphological dictionaries for German ([7]) and Dutch ([1]). The German morphological dictionary by Sabine Lehmann contains 3,977,448 entries. The automaton for the version with coded suffixes had 307,799 states (of which 62,790 formed chains), and 436,650 transitions. The version with coded suffixes, prefixes, and infixes had 107,572 states (of which 14,213 formed chains), and 176,421 transitions.

3.2 Results

Table 1 gives the size of automata built with various options. The *sg* automata with *SNTO*, *SNMT*, and *SNMTO* are bigger than expected because there was no space for one more flag in the pointer.

Table 1. Size of automata built with various options, Sabine Lehmann's German morphology , and CELEX Dutch morphology, in bytes. In the table, *sg* means Sabine Lehmann's German morphology with coded suffixes, *cd* – CELEX Dutch, *i* – coded prefixes and infixes, *O* – shared transitions, *S* - stop bit (lists of transitions), *N* – next pointers, *T* – tails of states, and *M* – changing the order of transitions.

		O	N	NM	NO	NMO
sg	2,178,268	2,117,628	1,747,210	1,733,730	1,706,512	1,694,405
sgi	882,123	822,133	731,713	718,439	692,514	681,007
cd	3,028,893	2,914,528	2,369,485	2,331,985	2,287,664	2,257,565
cdi	2,873,143	2,758,473	2,255,753	2,221,047	2,183,489	2,147,982

	S	SO	SMO	SN	SNM	SNO
sg	1,742,616	1,720,460	1,703,376	1,311,558	1,298,078	1,300,592
sgi	705,700	682,932	668,084	555,290	542,016	544,696
cd	2,423,116	2,382,936	2,350,584	1,763,708	1,726,208	1,737,200
cdi	2,298,516	2,257,728	2,225,540	1,681,126	1,646,420	1,663,904

	SNMO	STO	STMO	SNTO	SNMT	SNMTO
sg	1,282,648	1,703,292	1,690,229	1,507,949	1,511,461	1,495,909
sgi	528,814	666,202	654,613	531,578	542,016	523,299
cd	1,697,742	2,939,005	2,904,985	1,985,335	1,983,531	1,959,747
cdi	1,619,334	2,779,723	2,745,299	1,902,887	1,894,999	1,876,403

3.3 Conclusions

In case of German morphology, we managed to compress the initial automaton more than fourfold. With coded infixes and prefixes, we compressed the input data more than 696 times. Gzip compressed the input data (with coded infixes and prefixes) to 16,342,908 bytes. All automata for given input data could be made smaller by using compression by over 40% (43.9% for Dutch). The smallest automaton we obtained could still be compressed with gzip by 27.77%. The best compression method for German turned out to be a good preparation of the input data. It gave savings from 57.66% to 65.02%. For Dutch, those savings were only 4.15-5.50%, as words with prefixes and infixes constituted 3.68% of data, and not 22.93% as in case of German. As predicted, elimination of transition counters gave 20% on average. The figure was higher (up to 25.13% for German, 25.98% for Dutch) when next pointers were also used, as counters took proportionally more space in transitions. The figure was lower (16.93%) when only transition sharing was in use, because distributing a state over two other states was no longer possible. For German, next pointers gave savings from 15.77% to 24.70%, i.e. within the predicted range (3.22% –28.26%). For Dutch: 20.84% – 34.51%. The savings were bigger when the stop bit option was used. Surprisingly, transition sharing is less effective (0.84% – 3.01% on sg, and 1.91% – 7.24% on sgi, 0.98% – 4.45% on Dutch), and works better on sgi because sg contains many chains of states. Compression of tails of states adds only 0.71% to 2.45% for German, and does not work for Dutch data because the additional bit for a flag crosses the byte boundary. Changing the order of transitions gives small results

(up to 2.92%). We managed to speed it up so that it takes a fraction of the construction time.

Acknowledgments. We express our gratitude to Sabine Lehmann for making her dictionary available to us.

References

1. R. H. Baayen, R. Piepenbrock, and H. *The CELEX Lexical Database (CD-ROM).* Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA, 1993, 1993.
2. Jan Daciuk. *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing.* PhD thesis, Technical University of Gdańsk, 1998.
3. Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.
4. George A. Kiraz. Compressed storage of sparse finite-state transducers. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31. 1997.
5. Tomasz Kowal towski, Cláudio L. Lucchesi, and Jorge Stolfi. Finite automata and efficient lexicon implementation. Technical Report IC-98-02, January 1998.
6. Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brasil, 1993.
7. Sabine Lehmann. Multext German morphology. ISSCO, Geneva, September 12 1996.
8. Claudio Lucchiesi and Tomasz Kowaltowski. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, Jan. 1993.
9. Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes.* PhD thesis, Institut Blaise Pascal, Paris, France, 1991. LITP 91.44.
10. Max Silberztein. *Finite-State Language Processing*, chapter The Lexical Analysis of Natural Languages, pages 175–203. MIT Press, 1997.
11. Max Silberztein. INTEX tutorial notes. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31. 1999.
12. Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

Computing Raster Images from Grid Picture Grammars

Frank Drewes[1⋆], Sigrid Ewert[2⋆⋆], Renate Klempien-Hinrichs[3⋆], and Hans-Jörg Kreowski[3⋆]

[1] Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden
drewes@cs.umu.se
[2] Department of Computer Science, University of the Witwatersrand, Johannesburg, Private Bag 3, 2050 Wits, South Africa
sigrid@cs.wits.ac.za
[3] University of Bremen, Department of Computer Science, P.O. Box 33 04 40, D-28334 Bremen, Germany
{rena,kreo}@informatik.uni-bremen.de

Abstract. While a 2-dimensional grid picture grammar may generate pictures (defined as subsets of the unit square) with arbitrarily small details, only a finite number of them can be made visible as raster images for any given raster. We present an algorithm based on bottom-up tree automata which computes the set of all raster images of the pictures generated by a given grid picture grammar.

1 Introduction

Picture-generating devices, such as iterated function systems, chain-code picture grammars, turtle geometry picture grammars, cellular automata, random context picture grammars and collage grammars, specify in general infinite picture sequences or languages. See, for example, [PJS92, CD93, MRW82, PL90, EvdW99, DK99]. Although they are intended and used for modelling visible phenomena of various kinds, due to their infinity additional prerequisites are required to make the generated pictures visible. Depending on the resolution of the chosen output medium, be it a display or a printer, the generated pictures must be transformed into raster images before we can have a look at them. In other words, a picture-generating system or grammar of the above kind specifies two languages. The first language contains the intended pictures, which are usually described as geometric objects consisting of points, lines, squares, or other such parts in the Euclidean space of dimension 2, 3, or higher. The second language

⋆ Partially supported by the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems), the ESPRIT Basic Research Working Group APPLIGRAPH (Applications of Graph Transformation), and the *Deutsche Forschungsgesellschaft (DFG)* under grant no. Kr-964/6-1

⋆⋆ This research was performed while the author was a postdoctoral fellow at the University of Bremen, funded by the National Research Foundation of South Africa.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 113–121, 2001.

consists of the corresponding raster images for a given raster. The question arises how the picture language and the raster image language are related to each other. Clearly, one may transform a single generated picture into its raster image using standard techniques. With such a transformation, the raster image language—which is always finite—can be computed from a finite number of appropriately chosen pictures of the original language. However, which choice is appropriate? Or vice versa, given a raster image and a picture-generating device, is the image associated with any of the generated pictures?

In this paper, we study the relation between picture languages and raster image languages for the class of grid picture grammars, which are a normal form of grid collage grammars [Dre96] and which coincide with random context picture grammars [EvdW99] if one ignores the context conditions and fixes a uniform grid size for all productions. Moreover, iterated function systems where the functions are similarities mapping the unit square onto some cell of the $k\times k$-grid ($k \geq 2$) can be seen as special cases of grid picture grammars. As the main result, we show that the raster image language for each $r\times r$-raster of the unit square can be constructed from the given grid picture grammar. The construction is based on bottom-up tree automata which get derivation trees as input and compute the required raster information.

The paper is organised in the following way. In Section 2, 2-dimensional grid picture grammars and their generated picture languages, called galleries, are defined. The raster images of these pictures are defined in Section 3, which also provides our main results. In the conclusion, we discuss various generalizations of our considerations. In order to comply with the space restrictions, details and elaborated examples had to be omitted. Interested readers can find them in the long version [DEKK00].

2 Two-Dimensional Grid Picture Grammars

Let the unit square be divided by an evenly spaced grid into k^2 squares, for some $k \geq 2$. A *production* of a (two-dimensional) grid picture grammar consists of a nonterminal symbol on the left-hand side and the square grid on the right-hand side, each of the k^2 small squares in the grid being either black or white or labelled with a nonterminal.

A *derivation* starts with the *initial nonterminal* placed in the unit square. Then productions are applied repeatedly until there is no nonterminal left, finally yielding a generated picture. A production is applied by choosing a square containing a nonterminal A and a production with left-hand side A. The nonterminal is then removed from the square and the square is subdivided into smaller black, white, and labelled squares according to the right-hand side of the chosen production. The set of all pictures generated in this manner constitutes the *generated gallery*.

Example 1. For $k = 2$, a production is depicted in Figure 1 on the left. On the right, one can see a direct derivation replacing the topmost nonterminal square.

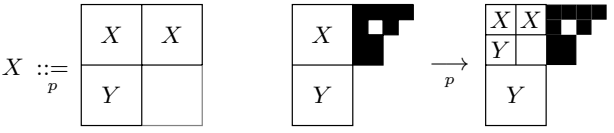


Fig. 1. A production of a 2×2-grid picture grammar and its application

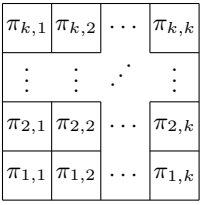


Fig. 2. Division of a square into k^2 subpictures

A picture generated by a grid picture grammar can be written as an expression in a convenient manner: Let the unit black square be represented by the symbol B and the corresponding white one by W. By definition, each of the remaining pictures in the generated language consists, as illustrated in Figure 2, of k^2 subpictures $\pi_{1,1}, \ldots, \pi_{1,k}, \ldots, \pi_{k,1}, \ldots, \pi_{k,k}$, each scaled by the factor $1/k$. If $t_{i,j}$ is the expression representing $\pi_{i,j}$ (for $i, j \in [k]$, where $[k]$ denotes the set $\{1, \ldots, k\}$), then $[t_{1,1}, \ldots, t_{1,k}, \ldots, t_{k,1}, \ldots, t_{k,k}]$ represents the picture itself (for $k = 2$ this is nothing else than a quadtree).

Formally speaking, any expression that represents a picture is a *term* (or *tree*) over a certain signature. In general, a *signature* is a finite set Σ of symbols, each symbol $f \in \Sigma$ being assigned a unique *rank* $\mathrm{rank}_\Sigma(f) \in \mathbb{N}$. The set T_Σ of *terms over* Σ is the smallest set such that for $n \in \mathbb{N}$, $f(t_1, \ldots, t_n) \in T_\Sigma$ for all $f \in \Sigma$ with $\mathrm{rank}_\Sigma(f) = n$ and all $t_1, \ldots, t_n \in T_\Sigma$. For $n = 0$, we may omit the parentheses in $f()$, writing just f instead.

Thus the terms that represent pictures in the k×k-grid are the terms in T_{Σ_k}, where $\Sigma_k = \{[-], \mathrm{B}, \mathrm{W}\}$, with $\mathrm{rank}_{\Sigma_k}([-]) = k^2$ and $\mathrm{rank}_{\Sigma_k}(\mathrm{B}) = \mathrm{rank}_{\Sigma_k}(\mathrm{W}) = 0$. As a notational convention, $[-](t_{1,1}, \ldots, t_{k,k})$ is written as $[t_{1,1}, \ldots, t_{k,k}]$.

It is now possible to define pictures and the relationship between pictures and terms in T_{Σ_k} formally. A *picture* π is simply a subset of the Euclidean plane, i.e., $\pi \subseteq \mathbb{R}^2$; intuitively, π is the set of black points. Each picture generated by a grid picture grammar is a subset of the unit square, i.e., $\pi \subseteq \mathrm{SQ} = \{(x, y) \in \mathbb{R}^2 \mid 0 \leq x, y \leq 1\}$. The *value* $\mathrm{val}(t)$ of a term $t \in T_{\Sigma_k}$ is the picture defined inductively by $\mathrm{val}(\mathrm{B}) = \mathrm{SQ}$, $\mathrm{val}(\mathrm{W}) = \emptyset$, and if $\mathrm{val}(t_{i,j}) = \pi_{i,j}$ for $i, j \in [k]$, then $\mathrm{val}([t_{1,1}, \ldots, t_{k,k}]) = [\![\pi_{1,1}, \ldots, \pi_{k,k}]\!]$, where

$$[\![\pi_{1,1}, \ldots, \pi_{k,k}]\!] = \bigcup_{i,j \in [k]} \mathrm{transform}^k_{i,j}(\pi_{i,j}),$$

and $\mathrm{transform}^k_{i,j}(\pi) = \left\{ \left(\frac{j-1+x}{k}, \frac{i-1+y}{k} \right) \mid (x, y) \in \pi \right\}$ for every picture π.

Hence, $\mathrm{val}([t_{1,1}, \ldots, t_{k,k}])$ is obtained by scaling and translating each picture $\pi_{i,j} = \mathrm{val}(t_{i,j})$ so that it fits into the (i, j)-th square of the k×k-grid.

Example 2. The term in T_{Σ_2} shown on the left of Figure 3 represents the picture on the right. For instance, the upper right quarter of the picture is described by the last line of the term.

$t = [\,[\mathrm{B}, [\mathrm{B}, \mathrm{B}, \mathrm{W}, \mathrm{B}], W, [\mathrm{B}, \mathrm{W}, \mathrm{B}, \mathrm{B}]\,],$
$\mathrm{W},$
$[\mathrm{B}, \mathrm{W}, [\mathrm{B}, \mathrm{W}, \mathrm{B}, \mathrm{B}], [\mathrm{B}, \mathrm{W}, \mathrm{B}, \mathrm{B}]\,],$
$[\mathrm{B}, \mathrm{W}, [\mathrm{B}, \mathrm{W}, \mathrm{B}, \mathrm{B}], [\mathrm{B}, \mathrm{W}, \mathrm{B}, \mathrm{B}]\,]]$

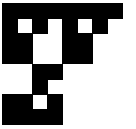


Fig. 3. A term $t \in T_{\Sigma_2}$ and its value $\mathrm{val}(t)$

Since there is a unique correspondence between terms and pictures, all we need in order to define grid picture grammars formally is an appropriate grammatical device for generating terms. Such a device is the regular tree grammar.

Definition 3. A *regular tree grammar* is a tuple $G = (N, \Sigma, P, S)$ consisting of a finite set N of *nonterminals* considered to be symbols of rank 0, a signature Σ disjoint with N, a finite set $P \subseteq N \times T_{\Sigma \cup N}$ of *productions*, and an *initial nonterminal* $S \in N$.

A term $t \in T_{\Sigma \cup N}$ *directly derives* a term $t' \in T_{\Sigma \cup N}$, denoted by $t \longrightarrow_P t'$, if there is a production $A ::= s$ in P such that t' is obtained from t by replacing an occurrence of A in t with s. The *language generated by* G is $L(G) = \{t \in T_\Sigma \mid S \longrightarrow_P^* t\}$, where $\longrightarrow_P^*$ denotes the reflexive and transitive closure of $\longrightarrow_P$; such a language is called a *regular tree language*.

Definition 4. A *grid picture grammar* is a regular tree grammar of the form $G = (N, \Sigma_k, P, S)$, for some $k \geq 2$. The *gallery generated by* G is $\Gamma(G) = \{\mathrm{val}(t) \mid t \in L(G)\}$.

Grid picture grammars are a normal form of 2-dimensional grid collage grammars, see [Dre96, Corollary 4.5]. The only difference is that grid collage grammars generate grid collages, which are sets of individual squares (i.e., are sets of sets of points), whereas here we are only interested in the resulting pictures (i.e., sets of points), obtained by taking the union of all squares in a given collage.

Strictly speaking, the formal definition of grid picture grammars is more general than the informal one discussed in the beginning, as it allows productions such as $A ::= [\mathrm{B}, [\mathrm{W}, A_1, A_2, \mathrm{W}], A_3, \mathrm{B}]$. Intuitively, on the right-hand side of this production the lower right subsquare is refined one level further than the other three. Productions of this kind are, however, not essential, due to the following well-known normal-form result on regular tree grammars (see, e.g., [GS97]).

Fact 5. *For every regular tree grammar G, a regular tree grammar $G' = (N, \Sigma, P, S)$ generating the same language can be constructed effectively such that P contains only productions of the form $A ::= f(A_1, \ldots, A_n)$, where $f \in \Sigma$, $\mathrm{rank}_\Sigma(f) = n$, and $A, A_1, \ldots, A_n \in N$.*

In the special case of grid picture grammars this means that only productions of the form $A ::= \mathrm{B}$, $A ::= \mathrm{W}$, and $A ::= [A_{1,1}, \ldots, A_{k,k}]$ are needed, where A and $A_{1,1}, \ldots, A_{k,k}$ are nonterminals.

Bottom-up tree automata can be seen as the inverse of regular tree grammars. Instead of generating a tree from the root to the leaves, they 'consume' an input tree, starting at the leaves and working upwards.

Definition 6. A *bottom-up tree automaton* is a tuple $ta = (\Sigma, Q, R, Q_f)$ consisting of an input signature Σ, a set Q of *states* which are considered to be symbols of rank 0, a set R of *rules* $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma$, $\mathrm{rank}_\Sigma(f) = n$, and $q_1, \ldots, q_n, q \in Q$, and a subset Q_f of Q, the set of *final states*.

For a term $t \in T_{\Sigma \cup Q}$, $t \longrightarrow_R t'$ if t' is obtained from t by replacing a subterm of t which is identical to the left-hand side of a rule in R with the right-hand side of that rule. The set $\mathrm{Acc}(ta)$ of *accepted terms* is given by $\mathrm{Acc}(ta) = \{t \in T_\Sigma \mid t \longrightarrow_R^* q \text{ for some } q \in Q_f\}$. A *finite bottom-up tree automaton* is a bottom-up tree automaton with only finitely many states.

Due to the following fact (see, e.g., [GS97]), finite bottom-up tree automata are a useful tool if one aims at computability results concerning sets generated by regular tree grammars.

Fact 7. *There is an algorithm which takes as input a regular tree grammar G and a finite bottom-up tree automaton ta, and decides whether*

$$L(G) \cap \mathrm{Acc}(ta) = \emptyset.$$

3 Computing Raster Images

A *raster*, such as the one provided by the finite resolution of a computer screen, can be seen as an $r \times r'$-grid that is evenly spaced in each direction. In order to simplify the situation, we shall consider in the following the idealised situation where $r = r'$ (and the area to be displayed is the unit square).

Given a picture $\pi \subseteq \mathrm{SQ}$, our aim is to determine the raster image that corresponds to π. The latter can be defined as follows: For a given square in the $r \times r$-grid, if any point of π lies properly inside the square, then that square is filled with the colour black, else it is coloured white. The *upper raster image* $\mathrm{raster}^{\mathrm{u}}_r(\pi)$ obtained in this way is the smallest set of raster squares which cover π, and thus the least upper bound on π with respect to the considered raster. More formally, let sq be SQ without its boundary, i.e., $\mathrm{sq} = \{(x, y) \in \mathbb{R}^2 \mid 0 < x, y < 1\}$. Then $\mathrm{raster}^{\mathrm{u}}_r(\pi) = \mathrm{val}([\mathrm{bw}^{\mathrm{u}}_{1,1}, \ldots, \mathrm{bw}^{\mathrm{u}}_{r,r}])$, where $\mathrm{bw}^{\mathrm{u}}_{i,j} = \mathrm{B}$ if $\pi \cap \mathrm{transform}^r_{i,j}(\mathrm{sq}) \neq \emptyset$ and $\mathrm{bw}^{\mathrm{u}}_{i,j} = \mathrm{W}$ otherwise.

Analogously, one may define the *lower raster image* of π as the largest raster image covered by π: $\mathrm{raster}^{\mathrm{l}}_r(\pi) = \mathrm{val}([\mathrm{bw}^{\mathrm{l}}_{1,1}, \ldots, \mathrm{bw}^{\mathrm{l}}_{r,r}])$, where $\mathrm{bw}^{\mathrm{l}}_{i,j} = \mathrm{B}$ if $\mathrm{transform}^r_{i,j}(\mathrm{sq}) \subseteq \pi$ and $\mathrm{bw}^{\mathrm{l}}_{i,j} = \mathrm{W}$ otherwise.

For a grid picture grammar G, the set of all upper $r \times r$-raster images obtained from pictures in $\Gamma(G)$ is denoted by $\mathcal{R}^{\mathrm{u}}_r(G) = \{\mathrm{raster}^{\mathrm{u}}_r(\pi) \mid \pi \in \Gamma(G)\}$. Similarly, $\mathcal{R}^{\mathrm{l}}_r(G) = \{\mathrm{raster}^{\mathrm{l}}_r(\pi) \mid \pi \in \Gamma(G)\}$ denotes the set of all lower raster images generated by G.

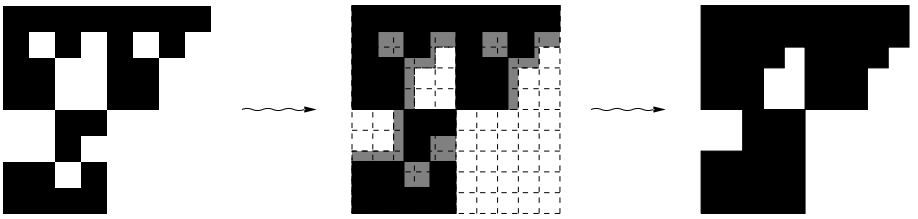


Fig. 4. Computing the upper raster image of a picture

Example 8. The picture considered in Example 2 'fits' exactly into an 8×8-grid. Figure 4 illustrates how its upper raster image is computed for a 10×10-raster: whenever the interior of a raster square is not completely white, it is painted black.

We shall now show that $\mathcal{R}_r^{\mathrm{u}}(G)$ is computable for every grid picture grammar G and every $r \in \mathbb{N}_+$ (where both G and r are given in the input). A corresponding result for $\mathcal{R}_r^{\mathrm{l}}(G)$ will then follow without much ado. For notational simplicity, let us fix an arbitrary grid size $k \geq 2$ to be used throughout the rest of this section. In particular, we denote $\mathrm{transform}_{i_1,i_2}^k$ by $\mathrm{transform}_{i_1,i_2}$.

Intuitively, if we consider a derivation in G, replacing all nonterminals in parallel in each step, the side length of the remaining nonterminal squares is k^{-n} after n steps. After finitely many steps all nonterminal squares have become smaller than the raster squares, so each of them is divided by at most one raster line horizontally respectively vertically. As depicted in Figure 5, this results in a segmentation of the nonterminal square into at most four rectangular subsets. If one of the dividing lines lies outside the square (or on one of its edges), or both of them do, two or even three of the rectangles are empty and are not considered.

Suppose that we continue the derivation. Upon termination, some of the four rectangles may contain points of the derived picture, turning the raster squares to which they belong black, while the remaining rectangles do not. Thus, situations such as the one shown in Figure 5 have to be investigated to develop the algorithm we aim at. In order to become independent of the actual size of the square under consideration, we magnify it to the size of the unit square SQ. Its segmentation into rectangles is uniquely determined by the positions of the correspond-

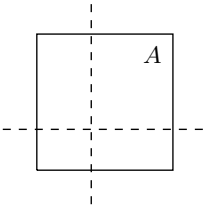


Fig. 5. Segmentation of a nonterminal square by raster lines

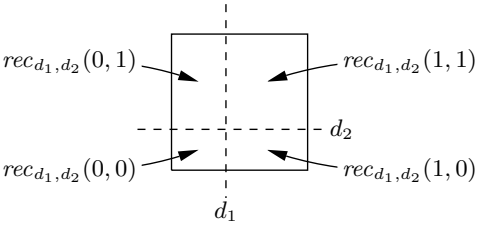


Fig. 6. Rectangular segments determined by d_1 and d_2

ing raster lines, two real coordinates. As described above, a pair $(d_1, d_2) \in \mathbb{R}^2$ of such coordinates divides the open square sq into at most four open rectangles. We can address these segments by the four corners $(c_1, c_2) \in \{0,1\}^2$ of SQ by defining $rec_{d_1,d_2}(c_1, c_2) = \{(x_1, x_2) \in \text{sq} \mid c_l < x_l < d_l \text{ or } d_l < x_l < c_l \text{ for } l \in \{1,2\}\}$, as illustrated in Figure 6. Now, let $\text{DIV}_{d_1,d_2} = \{rec_{d_1,d_2}(c_1, c_2) \mid c_1, c_2 \in \{0,1\}\}$. Given d_1, d_2 and a picture π, we are interested in those $rec \in \text{DIV}_{d_1,d_2}$ which intersect with π. The corresponding subset of DIV_{d_1,d_2} is denoted by $\text{div}_{d_1,d_2}(\pi)$: $\text{div}_{d_1,d_2}(\pi) = \{rec \in \text{DIV}_{d_1,d_2} \mid rec \cap \pi \neq \emptyset\}$.

The key to our algorithm is the lemma below. It states that $\text{div}_{d_1,d_2}(\pi)$ can be determined by a bottom-up tree automaton which processes the term denoting π. The automaton is finite if d_1, d_2 are rational numbers, because, roughly speaking, its states are the suffixes of the representation of d_1 and d_2 to the base k.

Lemma 9. *Let d_1, d_2 be rational numbers and $D \subseteq \text{DIV}_{d_1,d_2}$. One can effectively construct a finite bottom-up tree automaton ta_D such that*

$$\text{Acc}(ta_D) = \{t \in \text{T}_{\Sigma_k} \mid \text{div}_{d_1,d_2}(\text{val}(t)) = D\}.$$

Using the previous lemma, we can now prove the main theorem of this paper.

Theorem 10. *There is an algorithm which takes as input a grid picture grammar G and a number $r \in \mathbb{N}_+$, and yields as output the set $\mathcal{R}^{\text{u}}_r(G)$.*

Proof sketch. Consider a grid picture grammar $G = (N, \Sigma_k, P, S)$ in normal form (see Fact 5). By applying productions to the right-hand sides of productions in all possible ways, G can be turned into a $k^2 \times k^2$-grid picture grammar which generates the same language. Since this process can be repeated as often as necessary, it follows that we can assume without loss of generality that $k > r$.

Now, let us consider $\mathcal{R}^{\text{u}}_r(G)$. Since every derivation of a term in $L(G)$ has the form $S \longrightarrow_P \text{B}$, $S \longrightarrow_P \text{W}$, or $S \longrightarrow_P [A_{1,1}, \ldots, A_{k,k}] \longrightarrow^*_P [t_{1,1}, \ldots, t_{k,k}]$, the following raster images (and only those) are contained in $\mathcal{R}^{\text{u}}_r(G)$: (a) SQ if P contains the production $S ::= \text{B}$, (b) $\emptyset$ if P contains the production $S ::= \text{W}$, and (c) all images of the form $\text{raster}^{\text{u}}_r(\pi)$, where $\pi = [\![\text{val}(t_{1,1}), \ldots, \text{val}(t_{k,k})]\!]$ for some production $S ::= [A_{1,1}, \ldots, A_{k,k}]$ in P and derivations $A_{i_1,i_2} \longrightarrow^*_P t_{i_1,i_2}$ $(i_1, i_2 \in [k])$.

The raster images resulting from the first two items are easy to compute. Therefore, consider a production $S ::= [A_{1,1}, \ldots, A_{k,k}]$, and let $\Gamma_{i_1,i_2} = \{\text{val}(t) \mid A_{i_1,i_2} \longrightarrow^*_P t, t \in \text{T}_{\Sigma_k}\}$ denote the gallery A_{i_1,i_2} derives. We have to show how the set $\{\text{raster}^{\text{u}}_r([\![\pi_{1,1}, \ldots, \pi_{k,k}]\!]) \mid \pi_{i_1,i_2} \in \Gamma_{i_1,i_2} \text{ for } i_1, i_2 \in [k]\}$ can be computed.

Consider some $i_1, i_2 \in [k]$ and let (e_1, e_2) be the pair of rational numbers representing the raster lines which cut the square (i_1, i_2) (where $e_j = 0$ if there is no such line). Now, let $(d_1, d_2) = \text{transform}^{-1}_{i_1,i_2}(e_1, e_2)$ and define $\mathcal{D}_{i_1,i_2} = \{\text{div}_{d_1,d_2}(\pi_{i_1,i_2}) \mid \pi_{i_1,i_2} \in \Gamma_{i_1,i_2}\}$. By Lemma 9, in connection with Fact 7, $\mathcal{D}_{i_1,i_2}$ is computable, since Γ_{i_1,i_2} is generated by the grid picture grammar $G_{i_1,i_2} = (N, \Sigma_k, P, A_{i_1,i_2})$ and $\mathcal{D}_{i_1,i_2} = \{D \in \text{DIV}_{d_1,d_2} \mid L(G_{i_1,i_2}) \cap \text{Acc}(ta_D) \neq \emptyset\}$, where ta_D is as in Lemma 9.

By the definition of div_{d_1,d_2} and the choice of d_1 and d_2, a raster image *img* has the form

$$\mathrm{raster}_r^{\mathrm{u}}([\![\pi_{1,1}, \ldots, \pi_{k,k}]\!])$$

with $\pi_{i_1,i_2} \in \Gamma_{i_1,i_2}$ $(i_1, i_2 \in [k])$ if and only if $img = \mathrm{raster}_r^{\mathrm{u}}([\![D_{1,1}, \ldots, D_{k,k}]\!])$ for some $D_{i_1,i_2} \in \mathcal{D}_{i_1,i_2}$ $(i_1, i_2 \in [k])$. Since the (finite!) sets $\mathcal{D}_{i_1,i_2}$ are computable, these raster images are computable as well. □

Theorem 10 easily carries over to lower raster images.

Theorem 11. *There is an algorithm which takes as input a grid picture grammar G and a number $r \in \mathbb{N}_+$, and yields as output the set $\mathcal{R}_r^{\mathrm{l}}(G)$.*

Proof sketch. The picture $\pi = \mathrm{val}(t)$ represented by a term $t \in \mathrm{T}_{\Sigma_k}$ can easily be inverted by turning white squares into black ones and vice versa. Thus, the grid picture grammar $G' = (N, \Sigma_k, P', S)$ obtained from $G = (N, \Sigma_k, P, S)$ by exchanging B and W in all right-hand sides of productions generates the gallery of inverted pictures. As the lower raster image of a picture in $\Gamma(G)$ is the inverted upper raster image of the inverted picture, $\mathcal{R}_r^{\mathrm{l}}(G)$ can be obtained by computing $\mathcal{R}_r^{\mathrm{u}}(G')$ and subsequently inverting the resulting raster images. □

As an immediate consequence, we get the following corollary.

Corollary 12. *The following problems are decidable.*

1. *Given as input a grid picture grammar G, a number $r \in \mathbb{N}_+$, and an $r\times r$-raster image img: Is $img \in \mathcal{R}_r^{\mathrm{u}}(G)$?*
2. *Given as input two grid picture grammars G, G' (possibly based on different grids) and a number $r \in \mathbb{N}_+$: Is $\mathcal{R}_r^{\mathrm{u}}(G) = \mathcal{R}_r^{\mathrm{u}}(G')$?*

The analogous statements for $\mathcal{R}_r^{\mathrm{l}}$ instead of $\mathcal{R}_r^{\mathrm{u}}$ hold as well.

4 Conclusion

In this paper, we have presented an algorithm that computes the set of $r\times r$-raster images of the pictures generated by a 2-dimensional grid picture grammar. It is easily seen that some restrictions are imposed for technical reasons only and can be relaxed without problems. Since our reasoning works for the two axes of the plane independently and is closed under affine transformation, the underlying unit square may be replaced by a rectangle, or even a parallelogram, and the $k\times k$-grid underlying the grid picture grammar by any $k\times k'$-grid, for $k, k' \geq 2$. Similarly, the $r\times r$-raster may be generalised to an $r\times r'$-raster, for $r, r' \geq 1$. In fact, as far as the raster is concerned, it may be worthwhile to note that only three of its properties are used in the proofs, namely that the raster lines be parallel with the axes, be determined by rational numbers, and be finite in number. Thus, one could use a non-uniform raster provided it does not violate any of these requirements. Clearly, our considerations remain also true if we deal

with grids in the Euclidean space of dimension d with $d \geq 3$, or if we deal with grammars where distinct productions need not use the same grid.

In contrast to that, it seems much more difficult to get similar results for other types of picture generating devices, such as random context picture grammars, collage grammars, or iterated function systems, and for other modes of rewriting such as table-driven parallel or context-sensitive rewriting.

Last but not least, one could investigate more complex rasterisation functions than raster$^{\mathrm{u}}$ and raster$^{\mathrm{l}}$. An interesting candidate would be the one which blackens a raster square whenever the corresponding picture occupies at least half the area of the square. One could also consider using grey-scale values to express the percentage of black in a raster square. To study questions like these remains a matter of future work.

References

[CD93] Karel Culik II and Simant Dube. Affine automata and related techniques for generation of complex images. *Theoretical Computer Science*, 116:373–398, 1993.

[DEKK00] Frank Drewes, Sigrid Ewert, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Computing raster images from grid picture grammars. Report 2/00, Universität Bremen, 2000.

[DK99] Frank Drewes and Hans-Jörg Kreowski. Picture generation by collage grammars. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 397–457. World Scientific, Singapore, 1999.

[Dre96] Frank Drewes. Language theoretic and algorithmic properties of d-dimensional collages and patterns in a grid. *Journal of Computer and System Sciences*, 53:33–60, 1996.

[EvdW99] Sigrid Ewert and Andries van der Walt. Random context picture grammars. *Publicationes Mathematicae (Debrecen)*, 54 (Supp):763–786, 1999.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer, Berlin, 1997.

[MRW82] Hermann A. Maurer, Grzegorz Rozenberg, and Emo Welzl. Using string languages to describe picture languages. *Information and Control*, 54:155–185, 1982.

[PJS92] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals.* New Frontiers of Science. Springer, New York, 1992.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer, 1990.

A Basis for Looping Extensions to Discriminating-Reverse Parsing

Jacques Farré[1] and José Fortes Gálvez[2]

[1] Laboratoire I3S, CNRS and Université de Nice-Sophia Antipolis
[2] Dept. de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria

Abstract. We present a noncanonical extension to the *Discriminating Reverse* parsing method, which accepts non-LR grammars. In cases of parsing conflict, actions are deferred and a mark is virtually pushed onto the parsing stack. Then, locally-canonical DR parsing resumes until sufficient right context is read to resolve the initial conflict. Marks code coverings of the right contexts that are compatible with the actions in conflict. A suboptimal solution for such a coding is proposed, which is computed from the DR automaton itself. The stack vocabulary is enlarged with the mark set, but no new state is added to the basic DR automaton. Moreover, conflict resolution basically uses the DR parser. The method determines at construction time whether all the conflicts can be resolved, and only produces deterministic parsers.

1 Introduction

When designing a grammar, e.g., for a programming or specification language, the natural process is to reflect the logical constructions of the language. Unfortunately, such a naturally designed grammar will give place to a number of conflicts when given to most available parser generators. But conflict resolution by the user is often unsafe, because it usually forces to transform the grammar in unintended ways, what may result in accepting a different language. Thus, conflict resolution and, in general, imposing grammar adaptation by the user should be avoided as much as possible.

One of the reasons for such limitations in the classes of grammars that practical parser generators accept is that they limit the lookahead length to one because of the combinatorial complexity from using longer lookahead windows —notably in parsers based on the LR method. Efforts have been done to design safe parser generators allowing larger classes of grammars by using unbounded lookahead, but either they allow to build only relatively restricted parsers [1,9], or they accept ambiguous grammars and produce nondeterministic parsers [11, 12,2]. But nondeterminism is unacceptable in many application areas. In this paper we present the basis for noncanonical extensions to the *discriminating reverse* (DR) method in order to produce deterministic parsers for grammars and languages not restricted to LRR [3].

The DR approach to bottom-up parsing [5,6,7] allows to build linear deterministic DR(k) parsers for the general class of LR(k) grammars. It largely

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 122–134, 2001.

avoids the state number explosion of the "direct" Knuth construction [8]. Thus, the resulting parsers are simpler, smaller, and their construction is faster. DR(k) parsers are particularly interesting as our base because, contrary to state-stack parsers, the decision process is local to a (typically small) symbol-stack suffix, what allows to naturally resume parsing after a conflict. Moreover, it is quite natural to compute from the DR construction the conflict right contexts to follow.

The organisation of the paper is as follows. Section 2 reviews the basic DR(0) construction. Section 3 introduces NDR parsing and Section 4 presents in detail our proposed solution. In Section 5 the parser generation and parsing algorithms are given, and the last section presents the conclusions.

Notation.
We shall follow in general the usual conventions, as those in [10].

We will sometimes note grammar rules as $A \xrightarrow{i} \alpha$, where i is the rule number, $2 \leq i \leq |P| + 1$. Grammars $G(V, T, P, S)$ are considered to be augmented to $G'(V', T', P', S')$ with the addition of the rule $S' \xrightarrow{1} \vdash S \dashv$, and are supposed to be reduced.

$\hat{\varepsilon}$ will be a distinguished symbol. Symbols in $\hat{V}' = V' \cup \{\hat{\varepsilon}\}$ are noted $\hat{X}$, and strings in $\hat{V}'^*$ are noted $\hat{\alpha}$. By convention, $\hat{\varepsilon} \Rightarrow \varepsilon$.

2 Construction of the DR(0) Automaton

In order to simplify the presentation, we will use the DR(0) construction as the basis to our extension.

The basic idea of the discriminating reverse approach is to decide next shift-reduce parsing action by means of a parsing stack suffix exploration beginning from the stack top. For this exploration, a DFA is built such that each state is associated to the set of actions which are compatible with the suffix read thus far. As early as next transition would go to a single-action state, that action can be decided without exploring deeper.

The construction of the DR(0) automaton shown here follows that shown in [7], with the introduction of minor changes useful to the noncanonical extension presented in Section 3. The construction follows the item-set model associated to each state of the automaton.

2.1 DR(0) Items

A DR(0) item $[i, A \rightarrow \alpha \bullet \beta]$ has the following meaning:

- The i indicates the parsing action associated to the item (rule number, or 0 for shift). A dot besides it permits to know whether the rightpart to reduce has already been completely scanned. It will not be written when its position has no relevance, or when its position does not change.

– The core part is some rule $A \xrightarrow{j} \alpha\beta$ with a dot that represents the current stack exploration point, i.e., if σ is the already scanned stack suffix, then $\beta \underset{rm}{\Rightarrow}^{*} \sigma x$.

2.2 Closure of an Item Set

The closure of an item set I_q is the minimal set Δ_0 such that

$$\Delta_0(I_q) = I_q \cup \{[\bullet i, A \to \alpha \bullet B\beta] \mid A \to \alpha B\beta \in P', [i, B \to \bullet\gamma] \in \Delta_0(I_q)\}.$$

Since stack exploration is reverse, the rightpart dot moves in items from right to left, and thus the closure makes the dot ascend in the parsing tree while it is at the left end of rightparts.

The closure only generates items with left-dotted action codes, in order to trace that rightparts to reduce have been completely scanned.

2.3 Initial State and Transition Function on States

Since exploration begins at the stack top, all actions are valid at initial state q_0. We have to consider all possible valid positions of the stack top: the right end of rules' rightparts for the corresponding reductions when they are about to be reduced, and all the possible occurrences of terminal symbols within rightparts when the corresponding action is to shift.

$$I_{q_0} = \Delta_0(\{[i\bullet, A \to \alpha\bullet] \mid A \xrightarrow{i} \alpha \in P'\} \cup \{[\bullet 0, B \to \beta \bullet a\beta'] \mid B \to \beta a\beta' \in P'\}).$$

Transition on the next (lower in the stack) symbol is simply computed by moving the dot one position to the left[1]:

$$\Delta(I_q, X) = \Delta_0(\{[i, A \to \alpha \bullet X\beta] \mid [i, A \to \alpha X \bullet \beta] \in I_q\}).$$

Each item set is associated to a single state, i.e., $I_{q'} = I_q$ implies $q' = q$. It is useful to extend the transition function to strings in V'^{*}:

$$\Delta(I_q, \varepsilon) = I_q \qquad \Delta(I_q, X\alpha) = \Delta(\Delta(I_q, \alpha), X).$$

2.4 Transition Function on Nodes

The underlying directed graph on items within the DR(0) automaton item sets represents their relation in actual derivation trees, according to their relationship by transition and closure. It will allow us to follow the paths corresponding to these trees in order to recover the necessary context to resolve the DR(0) conflicts.

[1] Here applies the convention that, when no action dot is represented, its position is the same in both sides.

Thus, a node $\nu = [i, A{\to}\alpha{\bullet}\beta]_q$ is associated to an item $[i, A{\to}\alpha{\bullet}\beta] \in I_q$. The same item in different item sets will be different nodes.

(Single) transitions are defined on $\hat{V}'$:

$$\delta([i, A{\to}\alpha{\bullet}\beta]_q, \hat{X}) = \begin{cases} \{[{\bullet}i, B{\to}\varphi{\bullet}A\psi]_q\} & \text{if } \alpha = \varepsilon \text{ and } \hat{X} = \hat{\varepsilon} \\ \{[i, A{\to}\alpha'{\bullet}\hat{X}\beta]_{q'} \mid I_{q'} = \Delta(I_q, \hat{X})\} & \text{if } \alpha = \alpha'\hat{X} \\ \emptyset & \text{otherwise,} \end{cases}$$

where the special symbol $\hat{\varepsilon}$ is used for closure-related transitions. We extend this function to strings in $\hat{V}'^*$:

$$\delta(\nu, \varepsilon) = \{\nu\} \qquad \delta(\nu, \hat{\alpha}\hat{X}) = \bigcup_{\nu' \in \delta(\nu, \hat{X})} \delta(\nu', \hat{\alpha}).$$

Transition sequences on $\hat{\varepsilon}^*$ correspond to the closure on item sets.

2.5 Conflict-Root Nodes

Let I^C be the set of DR(0) conflict nodes, for which the left contexts indicated by the nodes (i.e., the leftpart nonterminal plus the rightpart portion to the left of the dot) are explicitly compatible with more than one parsing action, excluding the nodes which are found before a rightpart can be fully verified. The subset I_0^C contains the *conflict-root* nodes, i.e., the "initial" nodes of I^C, where conflicts are found. Hence,

$$I^C = \{\nu \mid \exists \hat{\rho}, \nu \in \delta([i, A{\to}\alpha X{\bullet}\alpha']_q, \hat{\rho}), \exists [i, A{\to}\alpha X{\bullet}\alpha']_q, [j, A{\to}\alpha X{\bullet}\alpha'']_q, \\ i \neq j, \nexists [k, B{\to}\beta X{\bullet}\beta']_q, B\beta \neq A\alpha, \nexists [h{\bullet}, A{\to}\alpha_1 W X{\bullet}\sigma]_q\}$$
$$I_0^C = \{\nu \mid \exists \nu' \notin I^C, \nu \in \delta(\nu', \hat{Y}) \cap I^C\}.$$

Note that the definition implies $\alpha' \neq \varepsilon$.

3 Introduction to Noncanonical DR(0) Parsing

In the NDR extension, in case of parsing conflict, actions are deferred, and a *conflict-root* mark is (virtually) pushed onto the parsing stack and a new shift is made. Then, the parser can resume its work as a normal DR parser (shifting, and reducing stack suffixes above the mark), until the mark position is reached during some stack suffix exploration. At that point, there are several possibilities, depending on the mark and the state in which it is found:

1. The conflict can be resolved, because the suffix leading to the state in which the mark is found is compatible with the right contexts associated with only one of the actions in conflict for this mark. At this point, the parser resumes using the conflict-root mark position as its new (virtual) stack top and performs the action.

2. The context associated to the mark allows to decide an action. It is performed (if it is not a reduction including the mark position) and normal parsing continues.
3. The parser cannot continue because the mark does not allow to decide between several actions, or because the reduction spans to the mark position. An *induced* mark is pushed, and DR parsing resumes as previously.
4. The mark is incompatible with the state (because the input is not a sentence). Therefore, an error is signalled.

Essential information regarding conflict contexts is computed at construction time. It is represented by graphs which can be deduced from the transitions from the initial state until the state in which the conflict is found. A graph is associated to each action in a DR conflict, and a mark is, at construction time, a set of graph nodes coding some covering of their corresponding right contexts to follow.

The critical question are the induced conflicts of point 3 above. An induced conflict can involve derivation subtrees, and the construction must leave some paths in the graph of the previous mark to follow the induced paths, and then resume the previous paths. In order to have optimal parsing power, a new graph must be connected to the node representing the corresponding mark position. Since this may happen an unbounded number of times, such a construction is, in principle, unfeasible, unless some form of looping construction is devised preserving the difference amongst the possible paths.

3.1 NDR(0) Parsing for an Example Non-LRR Language

In order to illustrate NDR(0) parsing, here we show by example the acceptation of a non-LRR (hence nondeterministic) language. Consider the grammar G_1 with productions

$$P_1' = \{\ S' \xrightarrow{1} \vdash S \dashv,\ S \xrightarrow{2} AC,\ S \xrightarrow{3} BCc,\ A \xrightarrow{4} aAb,\ A \xrightarrow{5} d,$$
$$B \xrightarrow{6} aBbb,\quad B \xrightarrow{7} d,\quad C \xrightarrow{8} aCc,\ C \xrightarrow{9} e\}$$

for the non-LRR[2] language $L_1 = \{a^n db^n a^p ec^p\} \cup \{a^n db^{2n} a^p ec^{p+1}\}$, $n, p \geq 0$.

The NDR(0) parsing table for the example grammar is shown in Fig. 1, where "q_i" represents transition entries, "i" represents a parsing action (rule number to reduce, or shift if $i = 0$), "m_i" indicates to push that mark, and "ri" indicates to resolve rightmost DR conflict by applying action i.

Note that mark situations are computed at construction time, but only mark themselves are used in coded form from the parsing table, as any other stack symbol.

Let us see, for instance, how parsing proceeds for a sentence in the first sublanguage of L_1 with an even number of b's (for an easier comprehension, marks

[2] Because a regular cover to always discriminate between suffix languages $b^n a^p ec^p$ and $b^{2n} a^p ec^{p+1}$ is impossible, for it would be necessary to compare the (unbounded) number of a's and c's.

	$\vdash$	$\dashv$	S	A	C	B	c	a	b	d	e	m_1	m_2
q_0	0	q_1	0	0	q_2	0	q_3	0	q_4	q_5	9		
q_1			q_{10}										r5
q_2				2		0		0				m_2	r5
q_3					q_9								r7
q_4				q_6		0			q_7			m_2	m_1
q_5	m_1							q_5					
q_6								4					
q_7						q_8						r7	
q_8								6					
q_9						3		8				r7	
q_{10}	1												

Fig. 1. NDR(0) parsing table for grammar G_1.

are shown as if they were actually inserted within the working sentential form). $\models$ and I represent a parsing step and the working top of stack, respectively.

$\vdash \mathsf{I} a^{2n}db^{2n}a^pec^p \dashv \stackrel{+}{\models} \vdash a^{2n}d\mathsf{I}b^{2n}a^pec^p \dashv \models \vdash a^{2n}d\,m_1b\mathsf{I}b^{2n-1}a^pec^p \dashv$
$\stackrel{+}{\models} \vdash a^{2n}d\,m_1(b\,m_2b\,m_1)^na\mathsf{I}a^{p-1}ec^p \dashv \stackrel{+}{\models} \vdash a^{2n}d\,m_1(b\,m_2b\,m_1)^na^pe\mathsf{I}c^p \dashv$
$\models \vdash a^{2n}d\,m_1(b\,m_2b\,m_1)^na^pC\mathsf{I}c^p \dashv \stackrel{+}{\models} \vdash a^{2n}d\,m_1(b\,m_2b\,m_1)^na^{p-1}C\mathsf{I}c^{p-1} \dashv$
$\stackrel{+}{\models} \vdash a^{2n}d\,m_1(b\,m_2b\,m_1)^nC\mathsf{I} \dashv \models \vdash a^{2n}d\,m_1(b\,m_2b\,m_1)^nC\,m_2 \dashv \mathsf{I}$
$\models \vdash a^{2n}A\mathsf{I}\,m_1(b\,m_2b\,m_1)^nC\,m_2 \dashv \models \vdash a^{2n}Ab\mathsf{I}\,m_2b\,m_1(b\,m_2b\,m_1)^{n-1}C\,m_2 \dashv$
$\models \vdash a^{2n-1}A\mathsf{I}\,m_2b\,m_1(b\,m_2b\,m_1)^{n-1}C\,m_2 \dashv \stackrel{+}{\models} \vdash A\mathsf{I}\,m_1C\,m_2 \dashv$
$\models \vdash AC\mathsf{I}\,m_2 \dashv \stackrel{+}{\models} \vdash S \dashv \mathsf{I}.$

The parser initially performs a sequence of shifts until finding d, whose possible reductions to A or B are in conflict, and thus conflict-root mark m_1 (i.e., the mark to resolve) is pushed and next terminal b is shifted. The following sequence of b's is shifted, with alternating marks m_2 and m_1 coding its parity. Then a^pec^p is reduced[3] to C without ever looking at the rightmost mark m_1. Then a new induced mark m_2 is pushed[4] and $\dashv$ symbol shifted, which allows to resolve the conflict-root mark in favor of reduction to A[5]. After reducing d to A, basically DR(0) parsing continues according to the new "stack top" position, reducing $a^{2n}db^{2n}$ to A. Then, a shift action puts C on top of the stack, and after some more steps parsing successfully ends.

The reader can easily check that the number of parsing actions and of stack explorations, including those regarding the conflict, is linear on the input length.

[3] This is the critical "counting" phase that is impossible for a regular covering. Note that there is neither special forward exploration nor backtracking, just natural locally-canonical DR(0) parsing.

[4] In case of an odd number of b's, the rightmost mark would be m_2, which would already resolve the reduction to A without shifting $\dashv$.

[5] Or, alternatively, to B if a c had been read instead of $\dashv$, for some sentential form $\vdash a^ndb^{2n}Cc \dashv$; note that the parser did not "count" the number of a's.

4 The Basic Looping Construction

In [4] a nonlooping solution is described, which directly uses the DR automaton underlying graph of items. We present here a more powerful, looping construction that computes more precise right contexts. Nevertheless, it is suboptimal since, in some cases, it merges paths of a same graph.

For this looping solution, contexts are coded by *mirror copies* of subgraphs of the DR automaton underlying graph. The nodes of these copies will be called *right* nodes, while DR automaton underlying graph nodes will be called *left* nodes. A distinct graph is built for each DR conflict. Each such DR conflict and graph can be uniquely identified by the pair (q, X), i.e., the DR conflict found at state q on symbol X.

4.1 Right Nodes and Transitions

A right node $\mu = [j, A{\to}\alpha_\bullet\beta, \nu_a, \nu_t]_{qX}$ belongs to the right context graph for action j in conflict (q, X). The core $A{\to}\alpha_\bullet\beta$ represents a potential mark situation where the dot indicates the stack top position at the moment of pushing a mark. Left node ν_a is the *reference* node of μ. Transitions for right nodes are guided by transition paths from ν_a towards left node ν_t. Accordingly, we define the following right-node transition function:

$$\theta([j, A{\to}\alpha_\bullet\beta, \nu_a, \nu_t]_{qX}, \hat{Y}) = \begin{cases} \hat{\theta}([j, A{\to}\alpha_\bullet\beta, \nu_a, \nu_t]_{qX}) & \text{if } \beta = \varepsilon \text{ and } \hat{Y} = \hat{\varepsilon} \\ \{[j, A{\to}\alpha\hat{Y}_\bullet\beta', \nu_a, \nu_t]_{qX}\} & \text{if } \beta = \hat{Y}\beta' \\ \emptyset & \text{otherwise,} \end{cases}$$

with[6]

$$\hat{\theta}([j, A{\to}\alpha_\bullet, \nu_a, \nu_t]_{qX}) = \begin{cases} \{[j, B{\to}\gamma A_\bullet\gamma', \nu'_a, \nu_t]_{qX} \mid \\ \quad \nu'_a = [i, B{\to}\gamma_\bullet A\gamma']_q \in \delta(\nu_a, \hat{\varepsilon}\alpha_1),\ \exists\, \hat{\rho}, \nu_t \in \delta(\nu'_a, \hat{\varepsilon}\hat{\rho})\} & \text{if } \nu_a \neq \hat{\nu} \\ \{[j, B{\to}\gamma A_\bullet\gamma', \hat{\nu}, \hat{\nu}]_{qX} \mid\ B{\to}\gamma A\gamma' \in P'\} & \text{otherwise.} \end{cases}$$

In right nodes, the dot moves from left to right, and when the right end of the rightpart is reached, it ascends in the parsing tree. Note the symmetry between the dot ascents in left and right context graphs:

$$[j, A{\to}\alpha B_\bullet\beta, \nu'_a, \nu_t]_{qX} \in \theta([j, B{\to}\varphi_\bullet, \nu_a, \nu_t]_{qX}, \hat{\varepsilon}) \text{ implies}$$
$$\nu'_a = [i, A{\to}\alpha_\bullet B\beta]_{q'} \in \delta([i, B{\to}_\bullet\varphi]_{q'}, \hat{\varepsilon}).$$

Transitions (on $\hat{\varepsilon}$) from right nodes with a reference node ν_a such that $\nu_t \in \delta(\nu_a, \hat{\varepsilon}\alpha_1)$ will be described in next sections.

[6] $\hat{\nu}$ is a distinguished virtual left node which indicates that all legal paths allowed by the grammar can be followed.

We extend θ to strings in $\hat{V}'^*$:

$$\theta(\mu, \varepsilon) = \{\mu\} \qquad \theta(\mu, \hat{X}\hat{\alpha}) = \bigcup_{\mu' \in \theta(\mu, \hat{X})} \theta(\mu', \hat{\alpha}).$$

4.2 Equivalence and Connection of Right Subgraphs

All right graphs such that their nodes have the same ν_t are mirror copies of the same section of the left graph defined by transitions from nodes in I_{q_0} towards ν_t. Hence, we use here a simple equivalence condition by considering that two right subgraphs are equivalent iff their nodes have the same (j, ν_t, q, X).

The following procedure connects the set of "target" nodes μ of a right subgraph associated with some (j, ν_t, q, X) to a μ_p of the right graph of the conflict (q, X) for action j.

procedure connect(ν_t, μ_p) $\triangleq$
  for all $\mu = [j, B{\to}\varphi_1\varphi_{2\bullet}, \nu'_a, \nu'_t]_{qX}$ **such that**
    $\mu_p = [j, A{\to}\alpha B_\bullet \beta, \nu_a, \nu_t]_{qX}$,
    $\nu'_t = [i, A{\to}\alpha_\bullet B\beta]_{q_t} \in \delta(\nu'_a, \hat{\varepsilon}\varphi_1), \nu'_a = [i, B{\to}\varphi_{1\bullet}\varphi_2]_{q_a}$
  add μ_p to $\theta(\mu, \hat{\varepsilon})$

Since two equivalent right subgraphs related to different left contexts are the same mirror copy of some left subgraph, their corresponding right contexts will be merged, and thus confused, because they share transitions built by *connect*. On the other hand, this property ensures looping on a right subgraph whose right context would infinitely reintroduce new mirror copies of the same subgraph in an unrestricted construction.

4.3 ε-Skip Function on Right Context Node Sets

We note by J_m the right node set for a mark m. The ε-skip function Θ_0 on mark node sets is defined as follows.

$$\begin{aligned}\Theta_0(J_m) = \{\mu' \mid \mu' = [j, A{\to}\alpha Y_\bullet \alpha', \nu_a, \nu_t]_{qX} \in \theta(\mu, \hat{\rho}),\ \mu \in J_m,\\ \hat{\rho} \Rightarrow^* \varepsilon,\ \exists [0, A{\to}\alpha Y_\bullet \alpha']_{q_0}\}\\ \cup \{\mu_0 \mid \mu_0 = [j, C{\to}\gamma Y_\bullet \gamma', \nu_0, \nu'_t]_{qX}, \nu_0 = [0, C{\to}\gamma Y_\bullet \gamma']_{q_0},\\ \nu'_t = [0, A{\to}\alpha_\bullet \alpha']_{q_t}, \mu_c = [j, A{\to}\alpha_\bullet \alpha', \nu_a, \nu_t]_{qX} \in \theta(\mu, \hat{\rho}),\\ \mu \in J_m,\ \nu'_t \in \delta(\nu_0, \hat{\varepsilon}\hat{\rho}'),\ \hat{\rho}\hat{\rho}' \Rightarrow^* \varepsilon\}.\end{aligned}$$

The objective of this function is to skip the nonreduced (because of the conflict giving place to the mark) nonterminals which can derive ε that the DR automaton cannot evidently find during its exploration of some working-stack suffix. Its first subset simply ascends on the right through θ, until a shift is possible; no connect is necessary in this case, and Θ_0 simply follows θ.

In its second subset, a connect is necessary, because it corresponds to the case where the DR automaton ascends in the parsing tree by δ through the

(nonexistent) stack suffix deriving ε, and finally deciding also a shift action — so, in all cases, the next parsing action after pushing a mark is to shift. In order to allow to resume the paths on the previous subgraphs, connections are performed (for each μ_c) by calls to $connect(\nu'_t, \mu_p)$, such that $\{\mu_p\} = \theta(\mu_c, B)$.

Note that it is possible to use a more precise equivalence condition which distinguishes whether a subgraph is associated with a closure, since they have in general a more restricted left context (deriving ε).

4.4 Transitions on a Mark

The computation of the induced conflict mark situations has the form of a transition function on the state q' finding the mark, as follows.

$$
\begin{aligned}
&\Theta(J_m, q') = \\
&\quad \Theta_0(\{\mu' \mid \mu' \in \theta(\mu, \beta), \mu = [j, A{\rightarrow}\alpha{\bullet}\beta\gamma, \nu_a, \nu_t]_{qX} \in J_m, I_{q'} = \Delta(I_{q_0}, \beta)\} \\
&\qquad \cup \{\mu_0 \mid \mu_0 = [j, D{\rightarrow}\varphi Y{\bullet}\varphi', \nu_0, \nu'_t]_{qX}, \nu_0 = [i, D{\rightarrow}\varphi Y{\bullet}\varphi']_{q_0}, \\
&\qquad\quad \mu_c = [j, A{\rightarrow}\alpha{\bullet}\beta B\gamma, \nu_a, \nu_t]_{qX} \in J_m, \\
&\qquad\quad \exists\, \hat{\rho}, [i, A{\rightarrow}\alpha{\bullet}\beta B\gamma]_{q'} \in \delta(\nu'_t, \beta) \subseteq \delta(\nu_0, \hat{\varepsilon}\hat{\rho})\})
\end{aligned}
$$

The first subset simply follows θ when the mark node is found during the exploration of a rightpart section from the initial state. On the other hand, for the second subset, as in Θ_0, when the DR automaton ascends in the trees through δ, "context switches" are performed, and the subgraphs are connected by calls to $connect(\nu'_t, \mu_p)$, such that $\{\mu_p\} = \theta(\mu_c, \beta B)$.

Each different node set is associated to a different mark, and thus $J_m = J_{m'}$ implies $m = m'$.

In general, for some mark m, not every possible mark m' will be produced at parsing time for every q such that $J_{m'} = \Theta(J_m, q) \neq \emptyset$. Such marks are useless, and their inclusion in the final parser could be avoided with a more precise generation procedure. Note that a mark set $J_{m'} = \Theta(J_m, q)$ is useless if some m_h is always found at parsing time before reaching m at q, and thus m' is never pushed. But, because of the equivalence condition, the context of $J_{m'}$ is never less precise than the context of any of such J_{m_h}. Therefore, the inclusion of useless marks does not affect the grammar class that the parser generator accepts.

4.5 DR(0)-Conflict Mark Node Sets

Let m_0^{qX} be the conflict-root mark associated with some DR(0) conflict (q, X). Its associated set of right nodes is the following:

$$J_{m_0^{qX}} = \Theta(\hat{J}_{qX}, q)$$

(what involves the calls to *connect* associated to Θ) where $\hat{J}_{qX}$ is the set associated to a virtual mark and is defined as follows:

$$\hat{J}_{qX} = \{[j, A{\rightarrow}\alpha X{\bullet}\beta, \hat{\nu}, \hat{\nu}]_{qX} \mid [j, A{\rightarrow}\alpha X{\bullet}\beta]_q \in I_0^C\}.$$

From the definition of I_0^C, the left contexts compatible with the actions in the conflict are the same. Thus, the graphs for actions in conflict are connected to a virtual right graph whose transitions follow all the upwards paths allowed by the grammar.

4.6 Inadequacy Condition

A grammar G is inadequate iff

$$\exists\, [j, A{\to}\alpha{\bullet}\beta, \nu_a, \nu_t]_{qX}, [j', A{\to}\alpha'{\bullet}\beta, \nu'_a, \nu'_t]_{qX} \in J_m,$$
$$j \neq j', (\nu'_a, \nu'_t) \in \{(\hat{\nu}, \hat{\nu}), (\nu_a, \nu_t)\}.$$

Since, from the node with $\hat{\nu}$, all legal paths allowed by the grammar can be followed, there is a path in each respective right graph of each action which follows the same transitions. Consequently, if the condition holds, there are some sentences for which the parser cannot discriminate between both such actions.

5 Algorithms

Here we present the NDR(0) parser-generation and parsing algorithms. The former produces the deterministic parsing table for the acceptable class of grammars that the latter uses during parsing. If no mark is computed, the grammar is LR(0) and the resulting table corresponds to a DR(0) parser, which may be considered a particular case of NDR(0).

5.1 NDR(0) Parser Generation

The generation algorithm iteratively generates automaton states and conflict marks. New states are created while an action cannot be discriminated, or if rightparts to reduce have not yet been fully explored. When conflict items are detected the corresponding conflict-root mark is pushed. In the mark section, actions to perform at the corresponding state when finding a mark are computed.

Inadequate grammars are detected according to the condition of previous section (not shown). If the grammar is adequate, a deterministic parsing table *Action* is generated.

NDR(0) GENERATOR:
  $Q := \{q_0\}$; $M := \emptyset$
  repeat
    for $q \in Q$ **do**
      for $X \in V'$ **do**
        $QS := \{(A\alpha, i) \mid \exists [i, A{\to}\alpha X{\bullet}\alpha']_q\}$
        if $QS = \emptyset$ **then** $Action(q, X) := error$
        else if $\exists i', \forall (A\alpha, i) \in QS, i = i'$ and $\nexists [i{\bullet}, C{\to}\gamma W X{\bullet}\gamma']_q$ **then**
            $Action(q, X) := i$

else if $\exists(A\alpha, i), (B\beta, j) \in QS, A\alpha \neq B\beta$ or $\exists[i_\bullet, C{\to}\gamma W X_\bullet\gamma']_q$ **then**
$I_{q'} := \Delta(I_q, X)$; add q' to Q; $Action(q, X) := goto\ q'$
else $J_m = J_{m_0^{qX}}$; add J_m to M; $Action(q, X) := push\ m$
until no new q
repeat
for $q' \in Q - \{q_0\}$ **do**
for $m \in M$ **do**
$MS := \{(j, i) \mid [j, A{\to}\alpha_\bullet\beta, \nu_a, \nu_t]_{qX} \in J_m, \exists[i, A{\to}\alpha_\bullet\beta]_{q'}\}$
if $MS = \emptyset$ **then** $Action(q', m) := error$
else if $\exists j', \forall(j, i) \in MS, j = j'$ **then** $Action(q', m) := resolve\ j'$
else if $\exists i', \forall(j, i) \in MS, i = i'$ and $\nexists\ [i'_\bullet, B{\to}\varphi X_\bullet\psi]_{q'}$ **then**
$Action(q', m) := i'$
else $J_{m'} := \Theta(J_m, q')$; add m' to M; $Action(q', m) := push\ m'$
until no new m

5.2 The NDR(0) Parser

In the following algorithm, a working list l contains the current symbol sequence deriving the input prefix read thus far. At any time, list positions can be indexed from 1 to the current length e. Procedure ins(X, p) inserts symbol X in position p in the list, and del(h, p) removes symbols at positions $p, \ldots, p + h - 1$. The parser uses pointer p to explore down the list from current "top" t.

Since sequences of simultaneous unresolved DR conflicts are resolved in the reverse order, we use a mark stack s. It contains one element per DR-conflict-root mark, with three components: the conflict-root mark position (and thus next resolution top) r, its corresponding rightmost mark position p —initially equal to r—, and rightmost mark value m. The topmost element, indexed by c, corresponds to the rightmost, current DR-conflict.

NDR(0) PARSER:
procedure shred(i) $\triangleq$ **case** i **of**
shift : **if** $t = e$ **then** read(a); ins($a,\ e + 1$)
$t := t + 1$
reduce($A \xrightarrow{i} \alpha$) : $t := t - |\alpha| + 1$; del($|\alpha|,\ t$); ins($A,\ t$)
$c := 1$; $s[c] :=(0,\ 0,\ 0)$
ins($\vdash,\ e + 1$); $t := e$
repeat
$q := q_0$; $p := t$
repeat
if $s[c].p \neq p$ **then** $act := Action(q,\ l[p])$
else $act := Action(q,\ s[c].m)$
case act **of**
goto q': $q := q'$; $p := p - 1$
shift, reduce : shred(act)
push m' : **if** $s[c].p \neq p$ **then** $c := c + 1$; $s[c].r := t$

$s[c].p := t$; $s[c].m := m'$; shred(*shift*)
resolve $i : t := s[c].r$; $c := c - 1$; shred(i)
until $act \neq goto$
until $act \in \{accept, error\}$

6 Conclusions

The basic construction of noncanonical discriminating-reverse parsers has been presented. It uses a mechanism of marks allowing to resume locally canonical DR parsing, while being able to resolve conflicts when sufficient right-hand context has been processed. This model of extension permits in general to largely increase the parsing power to non-LR(k) grammars while producing deterministic parsers, and it will thus be of applicability to problems where ambiguity or nondeterminism during parsing is hardly acceptable, as the area of programming language processing.

We conjecture that the optimal, finite construction without enlarging the basic DR(0) automaton will allow to parse at least every LALR(k) grammar, for any k, (since in those grammars, the left context information is condensed in a basically direct LR(0) parser, and the corresponding lookahead window is correlated to those states) as well as some class of grammars of infinite lookahead. Such an optimal solution is under study. Moreover, we have shown that, since the NDR parser retains all its context-free parsing power during the mark resolution phase, it is possible to accept non-LRR grammars and languages.

The likely straightforward extension to using DR(k) as the base automaton would evidently accept all LR(k) grammars with a relatively small increase in the number of states. Moreover, it would in general differentiate more left context in the automaton nodes, and thus the class of acceptable grammars would be further enlarged.

References

1. Boullier, P.: Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques. PhD Thesis, Université d'Orléans, France (1984)
2. Colmerauer, A.: Total precedence relations. Journal of the ACM, **17**:1 (1970) 14–30
3. Čulik II, K., Cohen, R.: LR-regular grammars. Journal of Computer and System Sciences **7** (1973) 66–96
4. Farré, J., Fortes Gálvez, J.: A Simple-Context Noncanonical Solution for Extended Discriminating-Reverse Parsing. I3S Technical Report TR-00-12 (2000)
5. Fortes Gálvez, J.: Generating LR(1) parsers of small size. In: Kastens, U., Pfahler, P. (eds): Compiler Construction, 4th International Conference, CC'92. Lecture Notes in Computer Science, Vol. 641, Springer-Verlag, Berlin (1992) 16–29
6. Fortes Gálvez, J.: A practical small LR parser with action decision through minimal stack suffix scanning. In: Dassow J. (ed.): Developments in Language Theory II. World Scientific, Singapour (1995)
7. Fortes Gálvez, J.: A Discriminating Reverse Approach to LR(k) Parsing. PhD thesis, Universidad de Las Palmas de Gran Canaria, Spain, and Université de Nice-Sophia Antipolis, France (1998)

8. Knuth, D. E.: On the translation of languages from left to right. Information and Control **8**:6 (1965) 607-639
9. Seité, B.: A Yacc extension for LRR grammar parsing. Theoretical Computer Science **52** (1987) 91–143
10. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, vol. I: languages and parsing. Springer-Verlag, Berlin (1988)
11. Tomita, M.: Efficient Parsing for Natural Language – A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Boston (1986)
12. Tomita, M. (ed.): Generalized LR Parsing. Kluwer Academic Publishers, Boston (1991)

Automata for Pro-V Topologies

Pierre-Cyrille Héam

LIAFA - Université Paris 7
case 7014, 2, place Jussieu
75251 Paris cedex 05
heam@liafa.jussieu.fr

Abstract. In this paper, we give an automata theoretic version of several algorithms dealing with profinite topologies. The profinite topology was first introduced for the free group by M. Hall, Jr. and by Reutenauer for the free monoid. It is the initial topology defined by all the monoid morphisms from the free monoid into a discrete finite group. For a variety of finite groups $\mathbf{V}$, the pro-$\mathbf{V}$ topology is defined in the same way by replacing "group" by "group in $\mathbf{V}$" in the definition. Recently, by a geometric approach, Steinberg developed an efficient algorithm to compute the closure, for some pro-$\mathbf{V}$ topologies (including the profinite one), of a rational language given by a finite automaton. In this paper we show that these algorithms can be obtained by an automata theoretic approach by using a result of Pin and Reutenauer. We also analyze precisely the complexity of these algorithms.

1 Preliminaries

For more informations on automata and languages theory we refer the reader to [4,7]. For a general reference on the theory of groups, the reader is referred to [19], and for basic results and definitions on graphs see [5].

1.1 Introduction

The profinite topology is used to characterize certain classes of rational languages: the languages of level 1/2 in the group hierarchy and the languages recognizable by reversible automata [11,14]. Moreover pro-$\mathbf{V}$ topologies on the free group or on the free monoid play a crucial role in the theory of finite semigroups [6,9,12]. In particular, several important decidability problems, related to the Malcev product, reduce to the computation of the closure of a rational language in the profinite topology.

Pin and Reutenauer [15,16] and Ribes and Zalesskii [17,18] developed an algorithm to compute the closure of a rational language given by a rational expression. Recently, by a geometric approach, Steinberg [20] proposed an algorithm to compute the closure of a rational language given by a finite state automaton.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 135–144, 2001.

This paper provides automata theoretic arguments to derive Steinberg's algorithm from Pin and Reutenauer's. Furthermore, we study several algorithmic consequences of this approach.

In the first part of this paper, we introduce some notations and definitions. In the second one, we give an algorithm to compute the subgroup of the free group generated by a rational language and we obtain an automata theoretic proof of Steinberg's algorithm. In the third part, we apply this algorithm to languages of the free monoid, and we show that testing whether a word belongs to the interior of a given rational language can be done in polynomial time. We also give a polynomial time algorithm to test whether a language is closed. In the fourth part we do a precise analysis of the algorithm in the profinite case. We deduce a new algorithm to test whether a deterministic automaton recognizes a closed language in the profinite topology (see [14,16]). Finally we give a new proof of a proposition on reversible automata (Proposition 10), and we also give a new characterization of closed rational subsets of the free group (Proposition 11).

1.2 Languages and Words

We denote by $|K|$ the cardinality of a set K. If φ is a function from a set X into a set Y and $x \in X$, we denote by $x\varphi$ the image of x by φ.

Let L be a language of A^*, we denote by L^c the complement of L in A^*.

Let A be a finite alphabet and let $\overline{A} = \{\overline{a} \mid a \in A\}$ be a copy of A. Finally, let $\tilde{A}$ be the disjoint union of A and $\overline{A}$. The map $a \mapsto \overline{a}$ from A onto $\overline{A}$ can be extended to a one-to-one function from $\tilde{A}$ into itself by setting $\overline{\overline{a}} = a$. A word of A^* is said to be reduced if it does not contain any factor of the form $a\overline{a}$ with $a \in \tilde{A}$. We denote by $D(A)$ the set of all reduced words of $\tilde{A}^*$ and by $\equiv$ the monoid congruence generated by the relations $a\overline{a} \equiv 1$ for all $a \in \tilde{A}$. We write $F(A)$ for the set $\tilde{A}/_{\equiv}$, which is a group for the quotient law, called the free group over A. Let π be the projection from $\tilde{A}$ into $F(A)$, which is a monoid morphism. Note that the restriction of π to $D(A)$ is one-to-one. For each element $u \in \tilde{A}$ there exists one and only one reduced word $v = \mathrm{red}(u)$ such that $u\pi = v\pi$. Since every word of A^* is reduced we can now identify $u \in A^*$ with $u\pi$, thereby considering A^* as a subset of $F(A)$.

1.3 Automata

Recall that a *finite automaton* is a 5-tuple $\mathcal{A} = (Q, B, E, I, F)$ where Q is a finite set of *states*, B is the alphabet, $E \subseteq Q \times B \times Q$ is the set of *edges* (or *transitions*), $I \subseteq Q$ is the set of *initial states* and $F \subseteq Q$ is the set of *final states*. A finite automaton $\mathcal{A} = (Q, B, E, I, F)$ is said to be a (n, m)-*automaton* if $|Q| = n$ and $|E| = m$. The following relation always holds for an (n, m)-automaton

$$m \leq |A|n^2 \text{ and hence } m = O(n^2)$$

A *path* in $\mathcal{A}$ is a finite sequence of consecutive edges:

$$p = (q_0, a_0, q_1), (q_1, a_1, q_2), \cdots, (q_{n-1}, a_n, q_n)$$

The *label* of the path p, denoted by $[p]$, is the word $a_1a_2\cdots a_n$, its *origin* is q_0 and its *end* is q_n. A word is accepted by $\mathcal{A}$ if it is the label of a path in $\mathcal{A}$ having its origin in I and its end in F. Such a path is said to be *successful*. The set of words accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

For every state q and language K we denote by $q._{\mathcal{A}}K$ (or $q.K$ if there is no ambiguity on $\mathcal{A}$), the subset of Q of all the states which are the end of a path having its origin in q and its label in K. An automaton is said to be *trim* if for each state q there exists a path from an initial state to q and a path from q to a final state. It is called *complete* if for each state p and each letter a of the alphabet there exists a transition which starts from p and which is labeled by a. An automaton is *deterministic* if it has a unique initial state and does not contain any pair of edges of the form (q, a, q_1) and (q, a, q_2) with $q_1 \neq q_2$. Let us note that for a deterministic (n, m)-automaton we have

$$m \leq |A|n \text{ and hence } m = O(n)$$

An important result of automata theory states that for any automaton $\mathcal{A}$ there exists exactly one deterministic automaton (up to isomorphism)with a minimal number of states which accepts the same language. It is called the *minimal automaton* of $L(\mathcal{A})$. An automaton is *connected* if any two states can be joined by a path. An *ε-automaton* is an automaton in which the transitions are in $Q \times (A \cup \{\varepsilon\}) \times Q$.

A subset L of $F(A)$ is said to be accepted by an automaton $\mathcal{A}$ on $\tilde{A}$ if $L(\mathcal{A})\pi = L$.

An automaton on $\tilde{A}$ is *dual* if for every transition (p, a, q), the triplet $(q, \overline{a}, p)$ also is a transition. An *inverse* automaton is a connected dual automaton on $\tilde{A}$ in which each letter induce a partial one-to-one function from the set of states into itself. We have [10,21]:

Proposition 1. *Let H be a subset of $F(A)$. The following propositions are equivalent:*

(i) H is a finitely generated subgroup of $F(A)$.
(ii) There exists an inverse automaton $\mathcal{A}$ on $\tilde{A}$ with a unique initial state which also is the unique final state such that $L(\mathcal{A})\pi = H$.
(iii) There exists a dual automaton $\mathcal{A}$ on $\tilde{A}$ with a unique initial state which also is the unique final state such that $L(\mathcal{A})\pi = H$.

We define a *subgroup automaton* as an inverse automaton with a unique initial state which also is the unique final one. The following result ([10,21]) describes an algorithm called **ReduceToInverse** to compute a subgroup automaton form a dual automaton.

Proposition 2. *Let $\mathcal{A} = (Q, \tilde{A}, E, \{i\}, \{i\})$ be a dual (n, m)-automaton. We can construct in time $O(mn + n^2)$ a subgroup automaton $\mathcal{A}_0$ on $\tilde{A}$ such that $L(\mathcal{A})\pi = L(\mathcal{A}_0)\pi$. Moreover $\mathcal{A}_0$ has no more states and transitions than $\mathcal{A}$.*

1.4 Automata and Graphs

Let $\mathcal{A}$ be a (n, m)-automaton with set of states Q and set of transitions E. A subset P of Q is said to be *strongly connected* if, for each pair p and q of states in P, there exist a path from p to q and a path from q to p. A strongly connected component of $\mathcal{A}$ is a maximal (for the inclusion) set of states which is strongly connected. The strongly connected components of $\mathcal{A}$ form a partition of Q. A transition (p, a, q) of $\mathcal{A}$ is *internal to a strongly connected component* if p and q belongs to the same strongly connected component. It is said *internal* if it is internal to some strongly connected component and *external* otherwise. One can compute in time $O(m + n)$ (see [5]) the strongly connected components of $\mathcal{A}$.

A spanning tree of a (n, m)-automaton $\mathcal{A} = (Q, B, E, I, F)$ is a spanning tree of the graph (E, Q) (see [5] for the definition of a spanning tree). It can be computed in time $O(m + n)$. Given a (n, m)-automaton, one can test whether a given word belongs to $L(\mathcal{A})$ in time $O((m + n)|u|)$. On the other hand, testing whether $L(\mathcal{A})$ is empty only requires time $O(m + n)$.

1.5 Rational Languages

The class of *rational* languages of M (where M is a monoid) is the smallest class of languages closed under product, finite union and star operation. It is well known that a language of A^* is rational if and only if it can be accepted by a finite automaton.

An analogous result holds in the free group, by a result of Benois [2].

Theorem 1. *A subset of $F(A)$ is rational if and only if it can be accepted by a finite automaton.*

1.6 Pro-V Topologies

A *variety* of finite groups is a class of finite groups closed under taking subgroups, homomorphic images and finite direct products. Important examples are $\mathbf{G}$, the variety of all finite groups; $\mathbf{G}_p$, the variety of all finite p-groups (where p is a prime number); $\mathbf{G}_{nil}$ and $\mathbf{G}_{sol}$, the varieties respectively of all finite nilpotent groups and all finite solvable groups; $\mathbf{G}_{com}$, the variety of all commutative finite groups.

We say that two elements x and y of $F(A)$ can be *separated* by a group V if there exists a group morphism $\varphi : F(A) \longrightarrow V$ such that $x\varphi \neq y\varphi$.

If for each group G and each normal subgroup H of G such that $H \in \mathbf{V}$ and $G/H \in \mathbf{V}$, we have $G \in \mathbf{V}$, then we say that $\mathbf{V}$ is *extension closed*. This is the case if $\mathbf{V} = \mathbf{G}$, $\mathbf{G}_p$ or $\mathbf{G}_{sol}$, but not if $\mathbf{V} = \mathbf{G}_{com}$ nor $\mathbf{G}_{nil}$. If $\mathbf{V}$ is extension closed and non trivial, then every pair of distinct elements of $F(A)$ can be separated by an element of $\mathbf{V}$. Thus we define

$$r_{\mathbf{V}}(x, y) = \min\{|V| \mid V \in \mathbf{V},\ V \text{ separates } x \text{ and } y\}$$

and

$$d_{\mathbf{V}}(x, y) = 2^{-r_{\mathbf{V}}(x,y)}$$

One can verify that $d_{\mathbf{V}}$ is a distance on $F(A)$ which defines the pro-**V** topology on $F(A)$. If $\mathbf{V} = \mathbf{G}$ we say profinite instead of pro-**G**. If K is a subset of $F(A)$ we denote by $Cl_{\mathbf{V}}(K)$ the closure of K for the pro-**V** topology.

Ribes and Zalesskii proved that if **V** is extension closed, we have

Proposition 3. [17,15,18] *One can compute the closure of a rational subset of $F(A)$ by the following algorithm:*

(1) $Cl_{\mathbf{V}}(S) = S$ *if* S *is finite.*
(2) $Cl_{\mathbf{V}}(S_1 \cup S_2) = Cl_{\mathbf{V}}(S_1) \cup Cl_{\mathbf{V}}(S_2)$.
(3) $Cl_{\mathbf{V}}(S_1S_2) = Cl_{\mathbf{V}}(S_1)Cl_{\mathbf{V}}(S_2)$.
(4) $Cl_{\mathbf{V}}(S^*) = Cl_{\mathbf{V}}(\langle S\rangle)$, *where* $\langle S\rangle$ *is the subgroup of* $F(A)$ *generated by* S.

Moreover if H is a finitely generated subgroup of $F(A)$ then $Cl_{\mathbf{V}}(H)$ also is a finitely generated subgroup of $F(A)$.

We also can define the pro-**V** topology on A^* as the trace on A^* of the pro-**V** topology on $F(A)$ (see [11]). So, to compute the closure in A^* of a rational set of A^* we just have to compute it in $F(A)$ and intersect the result with A^*.

In this paper we only consider extension closed varieties of groups. Moreover we assume that if H is a finitely generated subgroup of $F(A)$ given by a subgroup automaton $\mathcal{A} = (Q, \tilde{A}, E, \{i\}, \{i\})$, we can compute a subgroup (n, m)-automaton $\mathcal{A}_0$ such that $Cl_{\mathbf{V}}(H) = L(\mathcal{A}_0)\pi$. We call this algorithm **ClosureGroup** and denote its time complexity by $f(n, m)$. Let us remark that a subgroup (n, m)-automaton satisfies $m \leq |A|n$, and thus one can write $f(n)$ for $f(n, m)$. Such algorithms are known for the profinite and pro-p topologies (see [10]). Moreover we naturally assume that f is an increasing function of n and that $f(n_1 + n_2) \geq f(n_1) + f(n_2)$.

2 Main Algorithms

In this section we adapt the algorithm of Proposition 3 to rational languages given by an automaton. We first treat the case of a group generated by a rational language.

Proposition 4. *Let L be a non empty rational language on $\tilde{A}$ given by a trim (n, m)-automaton $\mathcal{A} = (Q, \tilde{A}, E, I, F)$. Let*

$$E_0 = E \cup \{(q, \overline{a}, p) \mid (p, a, q) \in E\} \cup \{(p, \varepsilon, q) \mid p, q \in F \cup I\}$$

Then the ε-automaton $\mathcal{A}_0 = (Q, \tilde{A}, E_0, I \cup F, F \cup I)$ satisfies

$$\langle L(\mathcal{A})\pi\rangle = L(\mathcal{A}_0)\pi$$

Proposition 5. *Let L be a rational language on $\tilde{A}$ given by a trim (n, m)-automaton $\mathcal{A} = (Q, \tilde{A}, E, I, F)$. One can compute in time $O(mn + n^2)$ an automaton $\mathcal{A}_0$ on $\tilde{A}$ such that $\langle L(\mathcal{A})\pi\rangle = L(\mathcal{A}_0)\pi$ and such that $\mathcal{A}_0$ is a subgroup automaton.*

Let us remark that thanks to the above result one can compute in polynomial time a finite set Y which generates the subgroup generated by $L(\mathcal{A})\pi$ (see [21]). It also is possible to compute such a set Y if L is presented by a rational expression, as it was presented in [1,4].

We deduce from this a method to compute the closure of a language given by a strongly connected automaton. Let us first recall the following proposition (see for example [11]):

Proposition 6. *Let $\mathcal{A} = (Q, \tilde{A}, E, I, F)$ be a trim dual automaton and for each $q \in Q$, let $H_q = L(\mathcal{A}^q)\pi$. The following propositions hold:*

(1) H_q is a finitely generated subgroup of $F(A)$.
(2) If $p \in q.u$ and $x = u\pi$ then $H_q = xH_p\overline{x}$.
(3) For each $i \in I$ and $f \in F$, let $u_{i,f}$ be a word on $\tilde{A}^*$ such that $f \in i.u_{i,f}$. Then:

$$L = \bigcup_{i \in I, f \in F} u_{i,f}\pi H_f = \bigcup_{i \in I, f \in F} H_i(u_{i,f}\pi) \tag{1}$$

Now we claim

Proposition 7. *Let $\mathcal{A}$ be a strongly connected-automaton on $\tilde{A}$ with a unique initial state i and a unique final state f. Let $H = L(\mathcal{A}^i)\pi$ and $L = L(\mathcal{A})\pi$. For all u and v in $\tilde{A}^*$ such that $f \in i.u$ and $i \in f.v$, we have*

$$\langle H \rangle = \langle L(v\pi) \rangle = \langle L(\overline{u}\pi) \rangle \tag{2}$$

and

$$Cl_{\mathbf{V}}(L) = Cl_{\mathbf{V}}(H)\overline{y} = Cl_{\mathbf{V}}(H)x = Cl_{\mathbf{V}}(\langle H \rangle)x = Cl_{\mathbf{V}}(\langle H \rangle)\overline{y} \tag{3}$$

Proposition 8. *Let $\mathcal{A} = (Q, \tilde{A}, E, I, F)$ be a strongly connected (n, m)-automaton. One can compute in time $O(f(n) + mn + n^2)$ an automaton $\mathcal{A}_0$ on $\tilde{A}$ such that $L(\mathcal{A}_0)\pi = Cl_{\mathbf{V}}(L(\mathcal{A})\pi)$*

Now we give an algorithm, called **ClosureInF**, to compute the closure of a rational language given by an automaton.

Theorem 2. *Let $\mathcal{A} = (Q, \tilde{A}, E, I, F)$ be an automaton. One can compute in time $O(f(n)+mn+n^2)$ an automaton $\mathcal{A}_0$ on $\tilde{A}$ such that $L(\mathcal{A}_0)\pi = Cl_{\mathbf{V}}(L(\mathcal{A}))\pi$. We call this algorithm **ClosureInF**.*

In particular pro-p and profinite closures of a rational set can be computed without computing a rational expression of it.

3 Applications to Languages of A^*

We use the previous algorithms to compute the closure of a rational set of A^* given by an automaton. We first need the following results.

Theorem 3. [2,3] *Let $\mathcal{A}$ be a (n, m)-automaton on $\tilde{A}$. One can compute in time $O(n^3)$ an automaton $\mathcal{A}_0$ on $\tilde{A}$ with n states such that $L(\mathcal{A})\pi = L(\mathcal{A}_0)\pi$ and if $u \in L(\mathcal{A})$ then red$(u) \in L(\mathcal{A})$.*

By retaining only the edges labeled by letters of A we get an important algorithm, called **ProjectionInA***.

Corollary 1. [3] *Let $\mathcal{A}$ be a (n, m)-automaton on $\tilde{A}$. One can compute in time $O(n^3)$ an n-state automaton which recognizes $L(\mathcal{A})\pi \cap A^*$.*

We can now sketch the description of our closure algorithm, called **ClosureInA***.

Theorem 4. *Let $\mathcal{A} = (Q, A, E, I, F)$ be an (n, m)-automaton. One can compute in time $O(f(n)^3)$ an automaton $\mathcal{A}_0$ on A which recognize the closure of $L(\mathcal{A})$ for the pro-$\mathbf{V}$ topology on A^*.*

If $\mathcal{A} = (Q, A, E, \{i\}, F)$ is a complete deterministic automaton then $L(\mathcal{A})^c$ also is a rational language accepted by the complete deterministic automaton $(Q, A, E, \{i\}, Q\backslash F)$. Thus it is easy to take the complement of a rational language of A^* given by a deterministic automaton. From this we deduce:

Proposition 9. *Let $\mathcal{A} = (Q, A, E, \{i\}, F)$ be a complete deterministic automaton and u a word on A. One can test in time $O(f(n)^3 + f(n)|u|)$ if u belongs to the interior of $L(\mathcal{A})$ for the pro-$\mathbf{V}$ topology on A^*.*

We again use the ease of taking the complement of a rational language of A^* given by a deterministic automaton. It is also easy to compute the intersection of two languages given by automata [7]. Thanks to this observation we have:

Theorem 5. *Let $\mathcal{A} = (Q, A, E, \{i\}, F)$ be a deterministic complete (n, m)-automaton. One can test in time $O(n^3 + (n + m)f(n))$ whether $L(\mathcal{A})$ is closed for the pro-$\mathbf{V}$ topology on A^*.*

4 The Profinite Case

In the case of $\mathbf{V} = \mathbf{G}$ we obtain the profinite topology and the algorithms become much easier. Indeed we can compute the closure of a language recognized by a strongly connected automaton just by taking its dual.

Theorem 6. *Let $\mathcal{A} = (Q, \tilde{A}, E, I, F)$ be an (n, m)-automaton on $\tilde{A}$. One can compute an automaton $\mathcal{A}_0$ such that $L(\mathcal{A}_0)\pi = Cl_{\mathbf{G}}(L(\mathcal{A})\pi)$ in time $O(m + n)$.*

We can also adapt it, as in the general case, to produce an algorithm to compute the profinite closure in A^*.

Theorem 7. *Let $\mathcal{A} = (Q, A, E, I, F)$ be a (n, m)-automaton on A. One can compute in time $O(n^3)$ an automaton $\mathcal{A}_0$ on A with n states which recognizes the closure of $L(\mathcal{A})$ for the profinite topology on A^*.*

We deduce from this a new algorithm to test whether a rational language of A^* is closed for the profinite topology on A^*. As for Theorem 5 we have:

Theorem 8. *Let $\mathcal{A} = (Q, A, E, I, F)$ be a deterministic (n, m)-automaton. One can test in time $O(n^3)$ whether $L(\mathcal{A})$ is closed for the profinite topology on A^*.*

In [14,16], the algorithm to test whether a language is closed for the profinite topology is obtained by checking a property of the ordered syntactic monoid. In practice it requires to compute the transitive closure of a graph with n^2 vertices and thus our algorithm is more efficient. Moreover this algorithm is useful to test whether a language is of level $1/2$ in the group hierarchy (see [14]) or whether it is recognizable by a reversible automaton (see [11]).

Moreover we obtain a new proof of the following proposition:

Proposition 10. [11] *A language of A^* which is accepted by a reversible automaton is closed for the profinite topology on A^*.*

As suggested in [20], we also have a new algorithm to compute the kernel of a finite monoid, which seems to have the same complexity as that developed in [13]. Let us first mention an important corollary of Theorem 7.

Corollary 2. [3] *Let $\mathcal{A}$ be a (n, m)-automaton on $\tilde{A}$ whose set of states is $Q = \{q_1, \cdots, q_n\}$. One can compute in time $O(n^3)$ a $n \times n$ boolean matrix such that the (i, j) entry is 1 if and only if there exists a path m in $\mathcal{A}$ from q_i to q_j such that $[m]\pi = 1$.*

Theorem 9. *Let M be a finite monoid generated by A. One can compute in time $O(|M|^3)$ the kernel of M given by its Cayley graph.*

Finally we have a new characterization of closed rational languages of the free group. An automaton $\mathcal{A}$ on $\tilde{A}$ is said to be *locally dual* if for each internal transition (p, a, q) of $\mathcal{A}$, $(q, \overline{a}, p)$ also is a transition of $\mathcal{A}$.

We claim that

Proposition 11. *L is a profinitely closed subset language of $F(A)$ if and only if there exists a locally dual automaton $\mathcal{A}$ on $\tilde{A}$ such that $L = L(\mathcal{A})\pi$.*

5 Conclusion

Our goal was to develop an algorithm to compute the closure of a rational language given by a finite state automaton. Using a result by Pin and Reutenauer we obtained such an efficient algorithm. However we do not know any lower bound for this problem. Furthermore our algorithm returns a non deterministic automaton, and we do not know whether, given a rational language given by a deterministic automaton, one can compute a deterministic automaton accepting the closure of the language in polynomial time or space.

The author would like to thank J.-E. Pin and P. Weil for fruitful comments.

References

1. Almeida, J.: Dynamics of implicit operations and tameness of pseudovarieties of groups. preprint. (2000)
2. Benois, M.: Parties rationnelles du groupe libre. C.R. Acad. Sci. Paris **269** (1969) 1188–1190
3. Benois, M., Sakarovitch, J.: On the complexity of some extended word problems defined by cancellation rules. Information Processing Letters **23** (1986) 281–287
4. Berstel, J.: Transductions and Context-Free-Languages. (1979)
5. Cormen, T.E., Leiserson, C.E., Rivest R.L.: Introduction to algorithms. MIT Press and McGraw-Hill Book Company. 6th edition (1992)
6. Delgado, M.: Abelian pointlikes of a monoid. Semigroup forum **56** (1986) 339–361
7. Eilenberg, S.: Automata, Languages and Machines. Academic Press. New York (1978)
8. Hall., M. A topology for free groups and related groups. Ann. of Maths. **52** (1950)
9. Henckell, K., Margolis, S., Pin, J.-E., Rhodes, J.: Ash's type II theorem, profinite topology and Malcev products. part I. Int. J. Algebra and Comput. **1** (1991) 411–436
10. Margolis, S., Sapir, M., Weil, P.: Closed subgroups in pro-**V** topologies and the extension problem for inverse automata. (to appear)
11. Pin. J.-E.: On the language accepted by finite reversible automata. Automata, Languages and Programming, 14th International Colloquium. Lecture Notes in Computer Science **267** (1987) 237–249
12. Pin, J.-E.: A topological approach to a conjecture of Rhodes. Bull. Austral. Math. Soc. **38** (1988) 421–431
13. Pin, J.-E.: Topologies for the free monoid. Journal of Algebra **137** (1991) 297–337
14. Pin, J.-E.: Polynomial closure of group languages and open sets of the Hall topology. Lecture Notes in Computer Science **820** (1994) 424–432
15. Pin, J.-E, Reutenauer, C.: A conjecture on the Hall topology on a free group. Bull. London Math. Soc. **25** (1991) 356–362
16. Pin, J.-E, Weil, P.: Polynomial closure and unambiguous product. Lecture Notes in Computer Science **944** (1995) 348–388
17. Ribes, L., Zalesskii, P.: On the profinite topology on a free group. Bull. London Math. Soc. **25** (1993) 37–43
18. Ribes, L., Zalesskii, P.: The pro-p topology of a free group and algorithmic problems in semigroups. Int. J. Algebra and Comput. **4** (1994) 359–374
19. Rotman, J.: An introduction to the theory of groups. Springer. New York. 4th edition (1995)

20. Steinberg, B.: Finite state automata: A geometric approach. Technical report. Univ. of Porto. (1999)
21. Weil, P.: Computing closures of finitely generated subgroups of the free group. Proceedings of the International Conference on Algorithmic Problems in Groups and Semigroups (2000) 289–307

Reachability and Safety in Queue Systems*

Oscar H. Ibarra

Department of Computer Science, University of California,
Santa Barbara, CA 93106, USA. ibarra@cs.ucsb.edu

Abstract. We look at a model of a queue system M that consists of the following components:

1. Two nondeterministic finite-state machines W and R, each augmented with finitely many reversal-bounded counters (thus, each counter can be incremented or decremented by 1 and tested for zero, but the number of alternations between nondecreasing mode and nonincreasing mode is bounded by a fixed constant). W or R (but not both) can also be equipped with an unrestricted pushdown stack.
2. One unrestricted queue that can be used to send messages from W (the "writer") to R (the "reader"). There is no bound on the length of the queue. When R tries to read from an empty queue, it receives an "empty-queue" signal. When this happens, R can continue doing other computation and can access the queue at a later time.

W and R operate at the same clock rate, i.e., each transition (instruction) takes one time unit. There is no central control. Note that since M is nondeterministic there are, in general, many computation paths starting from a given initial configuration. We investigate the decidable properties of queue systems. For example, we show that it is decidable to determine, given a system M, whether there is some computation in which R attempts to read from an empty queue. Other verification problems that we show solvable include (binary, forward, and backward) reachability, safety, invariance, etc. We also consider some reachability questions concerning machines operating in parallel.

1 Introduction

It is well-known that, in general, verification problems for infinite-state systems are undecidable [Esp97]. In fact, even for systems with only two variables (or counters) that can be incremented or decremented by 1 and tested for 0, we already know that the halting problem is undecidable [Min61]; hence, the emptiness, reachability, safety, and other problems are also undecidable. However, certain restrictions can be placed on the workings of these systems that make them amenable to analysis. Some models that have been shown to have decidable properties are: pushdown automata [BEM97,FWW97,Wal96], timed automata [AD94] (and real-time logics [AH94,ACD93,HNSY94]), and various approximations on multicounter machines [CJ98,BW94].

* Supported in part by NSF grant IRI-9700370.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 145–156, 2001.

This paper is a contribution to the reachability and safety analysis of infinite-state transition systems that can be modeled by a queue system. The model that we analyze, call it QS, is system M that consists of the following components:

1. Two nondeterministic finite-state machines W and R, each augmented with finitely many reversal-bounded counters (thus, each counter can be incremented or decremented by 1 and tested for zero, but the number of alternations between nondecreasing mode and nonincreasing mode is bounded by a fixed constant). W or R (but not both) can also be equipped with an unrestricted pushdown stack.
2. One unrestricted queue Q that can be used to send messages from W (the "writer") to R (the "reader"). Thus, from time-to-time during the computation, W can nondeterministically write a symbol in Q, and R can nondeterministically read a symbol from Q. There is no bound on the length of the queue. When R tries to read from an empty queue, it receives an "empty-queue" signal. When this happens, R can continue doing other computation and can access the queue at a later time.

W and R operate at the same clock rate, i.e., each transition (instruction) takes one time unit. There is no central control. Note that since M is nondeterministic there are, in general, many computation paths starting from a given initial configuration.

We investigate the decidable properties of queue systems. For example, we show that it is decidable to determine, given a system M, whether there is some computation in which R attempts to read from an empty queue. Other verification problems that we show solvable include (binary, forward, and backward) reachability, safety, invariance, etc. We also consider some reachability questions concerning machines operating in parallel.

A different model, where there is only one finite-state machine that controls the operation of the queue, can simulate a Turing machine (even when the machine has no counters). A restricted version of this model was recently studied in [IBS00]. The restriction is that the number of alternations between non-reading phase and non-writing phase is bounded by a constant. A non-reading (non-writing) phase is a period consisting of writing (reading) and no-changes, i.e., the queue is idle. For such systems, reachability and safety are decidable. Decidable properties of other varieties of systems with queues have also been been studied (see, e.g., [BH99]).

The paper has four sections, in addition to this section. Section 2 recalls the definition of pushdown automata with reversal-bounded counters (first introduced in [Iba78]) and cites some results we use in the paper. Section 3 presents the main results. Section 4 considers some reachability questions concerning machines operating in parallel. Section 5 is a brief conclusion.

2 Pushdown Automata with Reversal-Bounded Counters

A pushdown automaton with reversal-bounded counters (PCA) [Iba78] is a nondeterministic one-way pushdown automaton augmented with k "reversal-

bounded" counters (for some k). The pushdown stack is unrestricted. Without loss of generality, we assume that the counters can only store nonnegative integers, since the finite-state control can remember the signs of the numbers. Though not necessary (since it is one-way), we assume, for convenience, that the one-way read-only input to the PCA has left and right delimiters. A PCA without a pushdown stack is called a CA. PCAs, even CAs, are quite powerful. They can recognize rather complex languages. Decidability/complexity results concerning PCAs (CAs) have been obtained in [Iba78,GI81]. Some of the results were used recently to show the decidability/complexity of some decision problems (containment, equivalence, disjointness, etc.) for database queries with linear constraints [IS99,ISB00].

There are several papers that investigate verification problems for pushdown automata (e.g., [BEM97,FWW97,Wal96]), counter machines under various restrictions (e.g., [CJ98,FS00,ISDBK00], and pushdown automata with restricted counters (e.g., [BER95,BH96,IBS00]).

A fundamental result in [Iba78] is the following:

Theorem 1. *The emptiness problem for PCAs (i.e., given a PCA M, is the language, $L(M)$, accepted by M empty?) is decidable.*

Remark 1: It has been shown in [GI81] that the emptiness problem for CAs is decidable in n^{ckr} time for some constant c, where n is the size of the machine, k is the number of counters, and r is the reversal-bound on each counter. We believe that a similar bound could be obtained for the case of PCAs. We will see that the decision questions (reachability, safety, etc.) investigated in this paper are reductions to the emptiness problem.

PCAs can be generalized to have multiple input tapes (one head/tape). Thus, a k-tape PCA M accepts a relation $L(M)$ of k-tuples of strings. A 1-tape PCA will simply be called a PCA. A k-tape CA is k-tape PCA without a stack.

Corollary 1. *The emptiness problem for multitape PCAs (hence, also multitape CAs) is decidable.*

Proof. We sketch the proof for the case $k = 2$. Let M be a 2-tape PCA. We may assume without loss of generality that the two tapes of M use disjoint input alphabets. We construct a PCA M' such that $L(M')$ is empty if and only if $L(M)$ is empty. The idea of the construction is as follows: If (x_1, x_2) is an input to M, then the input to M' is a string x which is some interlacing of the symbols in x_1 and x_2. Thus x with the symbols in x_1 (x_2) deleted reduces to x_2 (x_1). Clearly M' can simulate the actions of the two input heads of M on input x. ∎

3 Main Results

We first look at a queue system with no pushdown stack. Let M be a QS with writer W, reader R, and queue Q. Both W and R can use instructions of the following form:

$q : x \leftarrow x + 1$ then goto p
$q : x \leftarrow x - 1$ then goto p
q : if $x\#0$ then goto p_1 else goto p_2
q : goto p
q : goto p_1 or goto p_2

where x represents a counter, $p, q, ...$ are states, and $\#$ is $<, =$, or $>$. We assume the states are labeled $\{1, 2, ...\}$. Without loss of generality (since the states can remember the sign), we assume that the counters can only take on nonnegative values. Hence, we can assume that $\#$ is $=$ (i.e., testing is only for checking whether a counter is zero). Note that the nondeterministic "goto p_1 or goto p_2" instruction is sufficient to simulate other types of nondeterminism. In addition, W can use instructions of the form:

q : write(a) then goto p

where a is a symbol in the queue alphabet. R, on the otherhand, can use instructions of the form:

q : if queue is empty then goto p else read/delete symbol and do T

where T is the form:

if symbol $= a_1$ then goto p_1 else
if symbol $= a_2$ then goto p_2 else
.....
if symbol $= a_n$ then goto p_n

where $a_1, ..., a_n$ are the symbols in the queue alphabet. "delete" means remove symbol from the queue.

Each counter in W and R is reversal-bounded in the sense that the number of times the counter changes mode from nondecreasing to nonincreasing and vice-versa is bounded by a constant, independent of the computation. So, for example,

0 1 1 2 3 3 3 4 5 5 5 4 3 2 1 1 0 0 1 1 2 3 3

is 2-reversal. We assume that each instruction takes one time unit and that the states of W (R) are labeled $1, 2, ...$

A configuration of M is a tuple $\alpha = (q, X, p, Y, w)$, where:

1. q is the state of R and X is a tuple of integer values of the counters of R.
2. p is the state of W, Y is a tuple of integer values of the counters of W, and w is the content (string of symbols) of Q.

Thus, a configuration represents the "total state" of the system at any given time. Note that α can be represented as a string where the components of the tuple are separated by markers and the states and counter values are written in unary.

Let $\alpha = (q, X, p, Y, w)$ and $\beta = (q', X', p', Y', w')$ be two configurations. We are interested in being able to check if β is reachable from α in 0 or more transitions. During the computation, when R attempts to access an empty queue, it receives an "empty-queue" signal. When this happens, R can continue doing other computation and can access the queue at a later time. We say that M is *non-blocking* with respect to a configuration α if M does not access an empty queue in *any* computation (i.e., sequence of moves) when started in configuration α. M is non-blocking if it is non-blocking with respect to any configuration α.

Theorem 2. *The following problems are decidable:*

1. *Given a QS M and a configuration α, is M non-blocking with respect to α?*
2. *Given a QS M, is it non-blocking?*

Proof. We only prove 2), the proof of 1) being similar. We show that given M, we can effectively construct a 2-tape CA M' such that $L(M')$ is empty if and only if M is non-blocking. Since the emptiness problem for multitape CAs is decidable (by Corollary 1), the result follows.

We describe the operation of M'. Let M' be given a pair of configurations (α, β) represented by $((q, X, p, Y, w), (q', X', p', Y', \epsilon))$, where (q, X, p, Y, w) is on tape1 and $(q,' X', p', Y', \epsilon)$ is on tape2. (Note that ϵ denotes an empty queue.) M' first reads q, X, p, Y from tape1 and q', X', p', Y' from tape2, and record these values. (M' uses auxiliary counters to record X, Y, X', Y'.) At this point, tape1 head is at the beginning of w and tape2 head is at ϵ. Then M' simulates the computation of R starting in state q and counter values X. M' uses a counter C_R to keep track of the running time of R. Immediately after R has read the last symbol of w on tape1, M' suspends the simulation (records the current values) and begins the simulation of W starting in state p and counter values Y. M' also uses a counter C_W to keep track of the running time of W. Immediately after W has written the first symbol, say a, on the queue (M' does not actually write a but records this symbol in the state), M' suspends the simulation of W and resumes the simulation of R until R reads a (which was recorded in the state). M' then resumes the simulation of W. The process of switching the simulations between W and R continues until M' "guesses" that W has written a symbol, say b, on a queue position after which blocking will occur. M' records the time t_W when b was written (this time is actually the current value of C_W). Then M' continues the simulation of R until R attempts to read past b. M' records the time t_R this happens. M' accepts if i) the states and counters of R and W after the simulations are the same as the corresponding values that M' recorded before the simulations, and ii) t_R is less than or equal to t_W (this condition indicates that M tried to access an empty queue). It follows that $L(M')$ is empty if and only if M is non-blocking. ∎

Remark 2: Theorem 2 does not hold for a QS model that has a second queue that can be used to send messages from R to W. It is easy to see that such a QS, even without counters, can simulate the computation of a Turing machine. A QS M guesses and verifies a halting sequence of instantaneous descriptions (IDs) of

the TM on blank tape: $ID_1\#ID_2\#ID_3\#...\#ID_k$. This is done as follows, where W_1 and R_1 belong to first machine of M and W_2 and R_2 belong to the second machine. The first machine operates as follows: W_1 writes ID_1 in the first queue and then iterates the following: R_1 reads ID_{i+1} from the second queue and W_1 writes ID_{i+2} on the first queue for $i = 1, 2, ...$ Similarly, the second machine iterates the following: R_2 reads ID_i from the first queue and W_2 writes ID_{i+1} in the second queue. It is obvious that the writing and reading on each queue can be appropriately scheduled so that the TM simulation can be carried out effectively. R is forced to access an empty queue if and only if the TM halts.

Remark 3: Similarly, Theorem 2 does not hold when there are two queues Q_1 and Q_2 that can be used to send messages from W to R (even when there are no counters), since such a system can simulate a TM. The idea is for W to nondeterministically guess and write a halting sequence of TM IDs $X_1\#X_2\#X_3\#...\#ID_k$ in Q_1 and a sequence of IDs $Y_1\#Y_2\#Y_3\#...\#Y_k$ in Q_2 such that Y_i is the successor of X_i. R reads Q_1 and Q_2 and checks that X_{i+1} is the successor of Y_i. R is then forced to access an empty queue if and only if $X_1\#Y_1\#X_2\#Y_2\#...\#X_k\#Y_k$ is a halting computation of a TM on blank tape.

W and R can be augmented with a pushdown stack and use instructions of the form:

q : if stack is empty then goto p else read/pop and do T

where T is of the form:

if top $= a_1$ then goto p_1 else
if top $= a_2$ then goto p_2 else
.....
if top $= a_n$ then goto p_n

where $a_1, ..., a_n$ are the symbols in the pushdown alphabet.

Corollary 2. *It is decidable to determine, given a QS M where R or W (but not both) has a pushdown stack, whether M is non-blocking.*

Proof. The definition of configuration α will now include the the pushdown content. Thus, $\alpha = (q, X, p, Y, w, z)$, where z is the tape content (with the rightmost symbol of z the top of the stack). The proof of Theorem 2 generalizes to a QS M with a pushdown stack. For example, if R has a stack, in the construction above, M' also writes z on its stack before simulating R and W. The rest of the construction is similar. M' also checks at the end of the simulations that the pushdown stack content of R corresponds to the z' portion of tape2. However, there is a slight complication because the pushdown stack content is in "reverse". If tape2 had z'^r (where r denotes reverse) instead of z', then there is no problem. One can get around this difficulty if the checking of the stack content with tape2 is done *during* the simulations instead of waiting until the end of the simulations. This involves guessing, for each position of the stack, the last time R rewrites

this position, i.e., that the symbol would not be rewritten further in reaching configuration β. So, e.g., if on stack position p, the symbol changes are $Z_1, ..., Z_k$ for the entire computation, then Z_k is the last symbol written on the position, and M' checks after Z_k is written that the p-th position of the stack word in β is Z_k. M' marks Z_k in the stack and makes sure that this symbol is never popped or rewritten in the rest of the computation. We omit the details. ∎

Remark 4: Corollary 2 does not hold when both R and W have a pushdown stack. In fact, it is undecidable to determine, given a QS M whose R and W have no counters but have each a one-turn pushdown stack (i.e., after popping, the stack can no longer push), whether M is non-blocking. The proof uses the undecidability of the Post Correspondence Problem (PCP). Let (X, Y) be an instance of the PCP, where $X = (x_1, ..., x_n)$ and $Y = (y_1, ..., y_n)$, each x_i, y_i a non-null string over the binary alphabet, $\{0, 1\}$. Let $a_1, ..., a_n, \#, \$$ be new symbols. We construct a QM M which operates as follows. W uses its one-turn pushdown stack to nondeterministically write in Q a string of the form:

$$a_{i_1} a_{i_2} ... a_{i_k} \# x^r_{i_k} ... x^r_{i_2} x^r_{i_1} \$$$

(Here, w^r denotes the reverse of w.) W writes the string symbol-by-symbol starting at time $t = 0$. After writing \$, W goes into an infinite loop without further writing on the queue. Starting at time $t = 1$, R reads the symbols on the queue pushing $a_{i_1} a_{i_2} ... a_{i_k}$ on its one-turn pushdown stack. When it sees #, R then checks if

$$x^r_{i_k} ... x^r_{i_2} x^r_{i_1} = y^r_{i_k} ... y^r_{i_2} y^r_{i_1}$$

If R finds a discrepancy, it goes into an infinite loop without further accessing the queue. If the check is successful, R reads the \$ and then attempts to read past this symbol. Thus, W will access an empty queue if and only if (X, Y) has a solution. The result follows since determining if an instance of the PCP has a solution is undecidable.

Remark 5: It is also undecidable to determine, given a QS M where R and W have no reversal-bounded counters but have each an unrestricted counter, whether M is non-blocking. We sketch the proof. Consider deterministic one-way finite-state acceptors augmented with one unrestricted counter. It is undecidable to determine, given two such acceptors, A_1 and A_2, whether they accept disjoint languages. In fact, we can assume that A_1 and A_2 are real-time (i.e., the input head moves right at every step) [HH70] (see also [Iba78]). We construct a QS M which operates as follows. Let # be a new symbol. Starting at time $t = 0$, W guesses and writes a string w (symbol-by-symbol) on the queue, while simultaneously checking if w is accepted by A_1. If w is accepted (not accepted) by A_1, R writes # (\$) and goes into an infinite loop without further writing on the queue. If A_1 gets "stuck" (i.e., there is no next move) at some symbol that W has written, W writes \$ and goes into an infinite loop without further writing on the queue. Thus the queue content will be a string of the form $w\#$ or of the form $w\$$. R starts reading the queue at time $t = 1$ and simulates A_2

on the symbols it reads. If A_2 accepts w and the symbol to the right of w is #, then R knows that w is accepted by both A_1 and A_2. Then R reads past #. Thus, R will access an empty queue. If w is not accepted by A_2 and the symbol to its right is # or \$, or if A_2 gets stuck while R is reading the queue, then R goes into infinite loop without further accessing the queue. It follows that M is non-blocking if and only if A_1 and A_2 do not accept a common string, which is undecidable.

Next, we look at reachability. If M is a QS, let $Reach(M)$ = set of all pairs of configurations (α, β) such that α can reach configuration β in 0 or more transitions.

Corollary 3. *We can effectively construct, given a non-blocking QS M where R or W (but not both) has a pushdown stack, a 2-tape PCA M' accepting $Reach(M)$.*

Proof. The construction of M' is similar to the one in Theorem 2 and Corollary 2. M' need only check that $C_R = C_W$ at the end of the simulation, knowing that during the simulation, R does not see the empty-queue marker. ∎

We are not able to prove the above corollary when M is unrestricted (i.e., may not be non-blocking) because for the simulation to proceed properly, we need to be able to tell whether or not the queue is empty every time R accesses the queue. However, we can prove the result when the QS has *no* pushdown stack.

Theorem 3. *We can effectively construct, given an unrestricted QS M with no pushdown pushdown stack, a 2-tape PCA M' accepting $Reach(M)$.*

Proof. The construction of M' involves two cases. The first case is when R reads only symbols from w and does not read any new symbol written by W during the computation from α to β. The second case is when R reads past w. M' begins by guessing which case to simulate. We describe only the operation of M' for the second case (the first one being similar). Like in the proof of Theorem 2, M' switches simulations between R and W but now uses a pushdown stack to tell whether R is accessing an empty Q. During the simulation, the stack keeps track of the difference in the time $t_R(j)$ the j^{th} symbol of x was read by R and the time $t_W(j)$ the j^{th} symbol of x was written by W. This is possible since the pushdown can be used as an unrestricted counter (i.e., there is no bound on the reversal). At some point during the simulation, after R has read the k^{th} symbol of x (for some k), M' guesses that R has read the last symbol it wants to read from the queue. M' continues the simulation of R and at some point guesses that R has reached the R-component of configuration β which M' can verify (this includes checking that the current state and counter values are equal to what were recorded before the start of the simulations). Then M' resumes the simulation of W including checking that the symbols written by the queue appear in tape2. The simulation continues until at some point M' guesses that W has reached the W-component of β which M' can verify. M' accepts if and only if $C_R = C_W$. ∎

In the rest of this section, we will consider only unrestricted QSs *without* a pushdown stack. The next result concerns nonsafety.

Theorem 4. *It is decidable to determine, given a QS M and two sets of configurations S (*start set*) and B (*unsafe set*) accepted by CAs, whether there is a configuration in S that can reach a configuration in B. Thus nonsafety is decidable.*

Proof. From Theorem 3, let M' be a 2-tape PCA accepting $Reach(M)$. Let M_S and M_B be the CAs accepting S and B, respectively. By using additional counters, we can modify M' to a 2-tape PCA M'' which also checks that α (β) on tape1 (tape2) is accepted by M_S (M_B). Then M is unsafe if and only if $L(M'')$ is nonempty, which is decidable by Corollary 1. ∎

Corollary 4. *It is decidable to determine, given a QS M and two sets of configurations S (*start set*) and G (*safe set*) accepted by a CA and a deterministic CA respectively, whether every configuration in S can only reach configurations in G. Thus invariance is decidable.*

Proof. It can be shown that if G is accepted by a deterministic CA M_G, then we can effectively construct a deterministic CA accepting the set of unsafe configurations B = the complement of G. (Note that this is not true if M_G is not deterministic.) The result follows from the above theorem. ∎

Next, we show that forward reachability is computable.

Theorem 5. *We can effectively construct, given a QS M and a set of configurations S accepted by a CA, a PCA accepting $post^*(M, S)$ = the set of all configurations reachable from configurations in S in 0 or more transitions.*

Proof. Let M_S be a CA accepting S. As in Theorem 4, we can construct a 2-tape PCA M' accepting the set of all pairs of configurations (α, β) in $Reach(M)$ such that α is accepted by M_S. We can then construct from M' a PCA M'' which deletes the first tape, and $L(M'') = post^*(M, S)$. ∎

Similarly, for forward reachability we have:

Corollary 5. *We can effectively construct, given a QS M and a set of configurations S accepted by a CA, a PCA accepting $pre^*(M, S)$ = the set of all configurations that can reach configurations in S in 0 or more transitions.*

We can define a QS acceptor M by giving R and W one-way input tapes. We say that a pair of strings (x, y) is accepted by M if on this pair of input strings and starting with all counters of R and W zero, M reaches a configuration when R and W are both in accepting states. We can show that the set of all pairs of strings accepted by a QS acceptor can be accepted by a 2-tape PCA.

If, however, the inputs to R and W are common, i.e., there is only one input tape from which R and W can read, the emptiness problem is undecidable (even when R and W are finite-state). The proof uses the idea in Remark 3.

4 Reachability in Parallel Machines

The technique of using the reversal-bounded counters to record and compare various integers (like the running times of the machines) in the proofs in the previous section can be used to decide some reachability questions concerning machines operating in parallel. The motivation for studying reachability is the following:

Suppose we are given a problem with input domain X, n nondeterministic machines $M_1, ..., M_n$, and an input x in X which can be partitioned into n components $x_1, ..., x_n$. Each machine M_i is to "work" on x_i to obtain a partial solution y_i. The solution to x can be derived from $y_1, ..., y_k$. We want to know if there is a computation (note that since the machines are nondeterministic, there may be several such computation) in which each M_i on input x_i outputs y_i and their running times t_i's satisfy a given linear relation (definable by a Presburger formula). An example of a relation is for each t_i to be within 5% of the average of the running times (i.e., the load is approximately balanced among the M_i's), or for the t_i's to satisfy some precedence constraints. Note that the t_i's need not be optimal as long as they satisfy the given linear relation. A stronger requirement is to find the optimal running time t_i of each P_i and determinine if $t_1, ..., t_n$ satisfy the given linear relation.

The questions above are unsolvable in general, even when the machines work independently (no sharing of data/variables). However, we are able to show that they are solvable for reversal-bounded multicounter machines with a pushdown stack. We illustrate below.

Let M_1 and M_2 be nondeterministic reversal-bounded multicounter machines with a pushdown stack but no input tape operating independently in parallel. Thus these machines are PCAs without input tape. Call them PCMs. For $i = 1, 2$, denote by α_i a configuration (q_i, X_i, w_i) of M_i (state, counter values, stack content). Let $L(m, n)$ be a linear relation definable by a Presburger formula. Define $Reach(M_1, M_2, L)$ to be the set of all 4-tuples $(\alpha_1, \beta_1, \alpha_2, \beta_2)$ such that for some t_1, t_2, M_i when started in configuration α_i can reach configuration β_i at time t_i, and t_1 and t_2 satisfy L, i.e., $L(t_1, t_2)$ is true. (Thus, e.g., if the linear relation is $t_1 = t_2$, then we want to determine if M_1 when started in configuration α_1 reaches β_1 at the same time that M_2 when started in α_2 reaches β_2.)

Theorem 6. *We can effectively construct, given two PCMs M_1 and M_2 and a Presburger relation L, a 4-tape PCA M accepting $Reach(M_1, M_2, L)$*

Proof. M, when given $\alpha_1, \beta_1, \alpha_2, \beta_2$ in its 4 tapes, first simulates the computation of M_1 to check that α_1 can reach β_1 recording the running time t_1 in a counter. M_2 then simulates M_2, recording the running time t_2 in another counter. Then M checks that t_1 and t_2 satisfies the given linear relation (which can be verified since Preburger formulas can be evaluated by nondeterministic reversal-bounded multicounter machines [Iba78]). ∎

We can allow the machines M_1 and M_2 to share common read-only data, i.e., each machine has a one-way read-only input head. A configuration α_i will

now be a 4-tuple (q_i, X_i, w_i, h_i), where h_i is the position of the input head on the common input x.

Suppose only one of M_1 and M_2 has a pushdown stack. The machine M to decide reachability is now a 5-tape PCA. The input 5-tuple is $(\alpha_1, \beta_1, \alpha_2, \beta_2, x)$. M simulates M_1 and M_2 in parallel on input x, recording their running times. If one of the machines, e.g., M_1 advances its input head to the next input symbol, but M_2 has not yet read the current input symbol, M does not advance its input head and "suspends"the simulation of M_1 until M_2 has read the current symbol or M guesses that M_2 will not be reading further on the input to reach its target configuration. We omit the details. Thus we have:

Corollary 6. *Reachability in two PCMs (only one of which has a pushdown stack) with a shared read-only input can be accepted by a 5-tape PCA.*

The above corollary is not true if both M_1 and M_2 have a one-turn stack (or an unrestricted counter), since reachability is undecidable, even if the machines have no reversal-bounded counters and the linear relation is $t_1 = t_2$. The proof is similar to the one given in Remark 4 (Remark 5). Similarly, if M_1 and M_2 share two common input tapes (i.e., each machine has two input heads for accessing the common tapes), reachability is undecidable, even if the machines are finite-state and the linear relation is $t_1 = t_2$. The proof is similar with Remark 3.

Note that the results above generalize to any number, k, of machines M_i $(i = 1, ..., k)$ operating in parallel.

5 Conclusions

We introduced a new model of a queue system and analyzed the solvability of verification problems such as (binary, forward, and backward) reachability, safety, and invariance. We also looked at reachability questions concerning machines operating in parallel. In the future we would like to investigate the decidability of liveness properties for these systems. We would also like to investigate the complexity of the verification procedures described in this paper. Finally, we mention an interesting open problem: Can we prove Theorem 3 when one of R or W (but not both) has a pushdown stack?

References

[ACD93] R. Alur, C. Courcoibetis, and D. Dill. Model-checking in dense real time. *Information and Computation*, 104(1):2-34, 1993.

[AD94] R. Alur and D. Dill. Automata for modeling real-time systems. *Theoretical Computer Science*, 126(2):183-236, 1994.

[AH94] R. Alur and T. A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181-204, 1994.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. *CONCUR 1997*, pp. 135-150.

[BG96] B. Boigelot and P. Godefroid. "Symbolic verification of communication protocols with infinite state spaces using QDDs." In *Proc. Int. Conf. on Computer Aided Verification*, pages 1–12, 1996.

[BER95] A. Bouajjani, R. Echahed and R. Robbana. "On the Automatic Verification of Systems with Continuous Variables and Unbounded Discrete Data Structures." In *Hybrid Systems II*, LNCS 999, 1995.

[BH96] A. Bouajjani and P. Habermehl. "Constrained Properties, Semilinear Systems, and Petri Nets." In *CONCUR'96*, LNCS 1119, 1996.

[BH99] A. Bouajjani and P. Habermehl. "Symbolic Reachability Analysis of FIFO-Channel Systems with Nonregular Sets of Configurations." *Theoretical Computer Science*,221(1-2): 211-250, 1999.

[BW94] B. Boigelot and P. Wolper, Symbolic verification with periodic sets, *Proc. 6th Int. Conf. on Computer Aided Verification*, 1994

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. *Proc. 10th Int. Conf. on Computer Aided Verification*, pp. 268-279, 1998.

[DIBKS00] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. Binary reachability analysis of discrete pushdown timed automata. To appear in *Int. Conf. on Computer Aided Verification, 2000.*

[Esp97] J. Esparza. Decidability of Model Checking for Infinite-State Concurrent Systems. *Acta Informatica*, 34(2): 85-107, 1997.

[FS00] A. Finkel and G. Sutre. Decidability of Reachability Problems for Classes of Two Counter Automata. *STACS'00.*

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *INFINITY, 1997.*

[GI81] E. M. Gurari and O. H. Ibarra. The complexity of decision problems for finite-turn multicounter machines. *JCSS*, 22: 220-229, 1981.

[HH70] J. Hartmanis and J. Hopcroft. What makes some language theory problems undecidable. *JCSS*, 4: 368-376, 1970.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Information and Computation*, 111(2):193-244, 1994.

[Iba78] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, Vol. 25, pp. 116-133, 1978.

[IBS00] O. H. Ibarra, T. Bultan, and J.Su. Reachability Analysis for Some Models of Infinite-state Transition Systems. To appear in *CONCUR'2000.*

[IS99] O. H. Ibarra and J. Su, A Technique for the Containment and Equivalence of Linear Constraint Queries. *Journal of Computer and System Sciences*, 59(1):1-28, 1999.

[ISB00] O. H. Ibarra, J. Su, and C. Bartzis, Counter Machines and the Safety and Disjointness Problems for Database Queries with Linear Constraints, to appear in *Words, Sequences, Languages: Where Computer Science, Biology and Linguistics Meet*, Kluwer, 2000.

[ISDBK00] O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. Kemmerer. Counter Machines: Decidable Properties and Applications to Verification Problems. To appear in *MFCS'2000.*

[Min61] M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. *Ann. of Math.*, 74:437-455, 1961.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Proc. Int. Conf. on Computer Aided Verification*, 1996

Generalizing the Discrete Timed Automaton*

Oscar H. Ibarra and Jianwen Su

Department of Computer Science, University of California,
Santa Barbara, CA 93106, USA. {ibarra, su}@cs.ucsb.edu

Abstract. We describe a general automata-theoretic approach for analyzing the verification problems (binary reachability, safety, etc.) of discrete timed automata augmented with various data structures. We give examples of such data structures and exhibit some new properties of discrete timed automata that can be verified. We also briefly consider reachability in discrete timed automata operating in parallel.

1 Introduction

Ever since the introduction of the model of a timed automaton [AD94], there have been many studies that extend the expressive power of the model (e.g. [BER95,CJ98,CJ99,DIBKS00,IDS00]). For instance [BER95] considers models of hybrid systems of finite automata supplied with (unbounded) discrete data structures and continuous variables and obtains decidability results for several classes of systems with control variables and observation variables. [CJ99,CJ98] shows that the binary reachability of timed automata is expressible in the additive theory of the reals. [DIBKS00] characterizes the binary reachability of discrete timed automata (i.e., timed automata with integer-valued clocks) augmented with a pushdown stack, while [IDS00] looks at queue-connected discrete timed automata.

In this paper, we extend the ideas in [DIBKS00,IDS00] and describe a general automata-theoretic approach for analyzing the verification problems of discrete timed automata augmented with various data structures. Formally, let $\mathcal{C}$ be a class of nondeterministic machines with reversal-bounded counters (i.e., each counter can be incremented or decremented by 1 and tested for zero, but the number of alternations between nondecreasing mode and nonincreasing mode is bounded by a constant, independent of the computation) and possibly other data structures, e.g., a pushdown stack, a queue, a read-write worktape, etc. Let A be a discrete timed automaton and M be a machine in $\mathcal{C}$. Denote by $A \oplus M$ the combined automaton, i.e., A augmented with M (in some precise sense to be defined). We show that if $\mathcal{C}$ has a decidable emptiness problem, then the (binary, forward, backward) reachability, safety, and invariance for $A \oplus M$ are also solvable. We give examples of such $\mathcal{C}$'s and exhibit some new properties of discrete timed automata that can be verified:

* Supported in part by NSF grant IRI-9700370.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 157–169, 2001.

1. For example, let A be a discrete timed automaton with k clocks. For a given computation of A, let r_i be the number of times clock i resets, $i = 1, ..., k$. Suppose we are interested in computations of A in which the r_i's satisfy a Presburger formula f, i.e., we are interested in the set Q of pairs of configurations (α, β) such that α can reach β in a computation in which the clock resets satisfy f. (A configuration of A is a pair (q, U), where q is a state and U is the set of clock values.) We can show that Q is Presburger. One can also put other constraints, like introducing a parameter t_i for each clock i, and consider computations where the first time i resets to zero is before (or after) time t_i. Then $Q(t_1, ..., t_k)$ is Presburger.
2. As another example, suppose we are interested in the set S of pairs of configurations (α, β) of a discrete timed automaton A such that there is a computation path (i.e., sequence of states) from α to β that satisfies a property that can be verified by a machine in a class $\mathcal{C}$. If $\mathcal{C}$ has a decidable emptiness problem, then S is effectively computable. For example, suppose that the property is for the path to contain three non-overlapping subpaths (i.e., segments of computation) which go through the same sequence of states, and the length of the subpath is no less than 1/5 of the length of the entire path. We can show that S is computable.

The constraints in 1 and 2 can be combined; thus, we can show that the set of pairs of configurations that are in both Q and S is computable.

3. We can equip the discrete timed automaton with one-way write-only tapes which the automaton can use to record certain information about the computation of the system (and perhaps even require that the strings appearing in these tapes satisfy some properties). Such systems can effectively be analyzed.

Finally, we briefly look at reachability in machines (i.e., $A_1 \oplus M_1$ and $A_2 \oplus M_2$) operating in parallel.

2 Combining Discrete Timed Automata with Other Machines

A timed automaton [AD94] is a finite-state machine augmented with finitely many real-valued clocks. All the clocks progress synchronously with rate 1, except that a clock can be reset to 0 at some transition. Here, we only consider integer-valued clocks. A *clock constraint* is a Boolean combination of *atomic clock constraints* in the following form: $x \# c$, $x - y \# c$, where $\#$ denotes $\leqslant, \geqslant, <, >$, or $=$, c is an integer, x, y are integer-valued clocks. Let $\mathcal{L}_X$ be the set of all clock constraints on clocks X. Let $\mathbb{Z}$ be the set of integers with $\mathbb{N}$ the set of nonnegative integers. Formally, a *discrete timed automaton* A is a tuple $\langle S, X, E \rangle$ where

1. S is a finite set of *(control) states*,
2. X is a finite set of *clocks* with values in $\mathbb{N}$, and
3. $E \subseteq S \times 2^X \times \mathcal{L}_X \times S$ is a finite set of *edges* or *transitions*.

Each edge $\langle s, \lambda, l, s' \rangle$ in E denotes a transition from state s to state s' with *enabling condition* $l \in \mathcal{L}_X$ and a set of clock resets $\lambda \subseteq X$. Note that λ may be empty. The meaning of a one-step transition along an edge $\langle s, \lambda, l, s' \rangle$ is as follows:

- The state changes from s to s'.
- Each clock changes. If there are no clock resets on the edge, i.e., $\lambda = \emptyset$, then each clock $x \in X$ progresses by one time unit. If $\lambda \neq \emptyset$, then each clock $x \in \lambda$ is reset to 0 while each $x \notin \lambda$ remains unchanged.
- The enabling condition l is satisfied.

The notion of a discrete timed automaton defined above is slightly different, but easily shown equivalent to the standard definition of a (discrete) timed automaton in [AD94] (see [DIBKS00]).

Now consider a class $\mathcal{C}$ of acceptors, where each machine M in the class is a nondeterministic finite automaton augmented with finitely many counters, and possibly other data structures. Thus, $M =< Q, \Sigma, q_0, F, K, D, \delta >$, where Q is the state set, Σ is the input alphabet, q_0 is the start state, F is the set of accepting states, K is the set of counters, D the other data structures, and δ is the transition function. In the move $\delta(q, a, s_1, ..., s_k, loc) = \{t_1, ..., t_m\}$,

- q is the state, a is ϵ or a symbol in Σ, s_i is the status of counter i (i.e., zero or non-zero), and *loc* is the "local" portion of the data structure(s) D that influences (affects) the move. For example, if D is a pushdown stack, then *loc* is the top of the stack; if D is a two-way read-write tape, then *loc* is the symbol under the read-write head; if D is a queue, then *loc* is the symbol in the front of the queue or ϵ if the queue is empty. Note that D can be a combination of several data structures (e.g., several stacks and queues).
- $t_1, ..., t_m$ are the choices of moves (note that M is nondeterministic). Each t_i is of the form $(p, d_1, ..., d_k, act)$, which means increment counter i by d_i (1, 0, or -1), perform *act* on *loc*, and enter state p. For example if D is a pushdown stack, *act* pops the top symbol and pushes a string (possibly empty) onto the stack; if D is a two-way read-write tape, *act* rewrites the symbol under the read-write head and moves the head one cell to the left, right, or remains on the current cell; if D is a queue, then *act* deletes *loc* (if not ϵ) from the front of the queue, and possibly add a symbol to the rear of the queue.

Note that the counters can only hold nonnegative integers. There is no loss of generality since the states can remember the signs.

The language accepted by M is denoted by $L(M)$. We will only be interested in $\mathcal{C}$'s with a decidable emptiness problem. This is the problem of deciding for a given acceptor in $\mathcal{C}$, whether $L(M)$ is empty. Since the emptiness problem for finite automata augmented with two counters is undecidable [M61], we will need to put some restrictions on the operation of the counters.

Let r be a nonnegative integer. We say that a counter is r-reversal if the counter changes mode from nondecreasing to nonincreasing and vice-versa at

most r times, independent of the computation. So, for example, a counter whose values change according the pattern 0 1 1 2 3 3 3 4 5 5 5 4 3 2 1 1 0 0 1 1 2 3 3 is 2-reversal. When we say that the counters are *reversal-bounded*, we mean that we are given an integer r such that each counter is r-reversal. From now on, we will assume that the acceptors in $\mathcal{C}$ have reversal-bounded counters.

We can extend the acceptors in $\mathcal{C}$ to *multitape* acceptors by providing them with multiple one-way read-only input tapes. Thus, a k-tape acceptor now accepts a k-tuple of words (strings). We call the resulting class of acceptors $\mathcal{C}(k)$. The emptiness problem for $\mathcal{C}(k)$ is deciding for a given k-tape acceptor M, whether it accepts an empty set of k-tuples of strings. We denote $\mathcal{C}(1)$ simply by $\mathcal{C}$. One can easily show the following:

Theorem 1. *If the emptiness problem for $\mathcal{C}$ is decidable, then the emptiness problem for $\mathcal{C}(k)$ is decidable.*

In the rest of the paper, we will assume that $\mathcal{C}$ has a decidable emptiness problem. In the area of verification, we are mostly interested in the "behavior" of machines rather than their language-accepting capabilities. When dealing with machines in $\mathcal{C}$ *without* inputs, we shall refer to them simply as machines. Thus, when we say "a machine M in $\mathcal{C}$", we mean that M has *no* input tape.

Let A be a discrete timed automaton and M a machine in class $\mathcal{C}$ (hence, M has no input tape!). Let $A \oplus M$ be the machine obtained by augmenting A with M. So, e.g., if M is a machine with a pushdown stack and reversal-bounded counters, then $A \oplus M$ will be a discrete pushdown timed automaton with reversal-bounded counters. We will describe more precisely how $A \oplus M$ operates later. A configuration α of $A \oplus M$ is a 5-tuple $(s, U, q, V, v(D))$, where s and U are the state and clock values of A, and q, V, $v(D)$ are the state, counter values, and data structure values of M (e.g., if D is a pushdown stack, then $v(D)$ is the content of the stack; if D is a queue, then $v(D)$ is the content of the queue). Let $Reach(A \oplus M)$ be the set of all pairs of configurations (α, β) such that α can reach β. This set is the binary reachability of $A \oplus M$. We assume that the configurations are represented as strings over some alphabet, where the components of a configuration are separated by markers and the clock and counter values represented in unary. We also assume that each of the following tasks can be implemented on a machine M' in $\mathcal{C}$: (i) M', when given a configuration $\alpha = (s, U, q, V, v(D))$ of $A \oplus M$ on its input tape, can represent this configuration in its counters and data structures, i.e., M' can read α and record the states s and q, store the set of values of U and V in appropriate counters, and store $v(D)$ in its data structures. (ii) M', when given a configuration α on its input tape, can check if α represents its current configuration (this task is the converse of (i)).

In the following, A is a discrete timed automaton and M is a machine in $\mathcal{C}$; FCA refers to a nondeterministic finite automaton (acceptor) augmented with reversal-bounded counters.

Theorem 2. *We can effectively construct a 2-tape acceptor in $\mathcal{C}(2)$ accepting $Reach(A \oplus M)$. Note that the input to the 2-tape acceptor is a pair of configurations (α, β), where α (β) is on the first (second) tape.*

We sketch the proof of the above theorem in the next section for a particular class $\mathcal{C}$. We omit the proofs of the next four theorems in this extended abstract.

Theorem 3. *If I (the initial set) and P (the unsafe set) are two sets of configurations of $A \oplus M$, let BAD be the set of all configurations in I that can reach configurations in P. If I and P can be accepted by FCAs, then we can effectively construct an acceptor in $\mathcal{C}$ accepting BAD. Hence, nonsafety is decidable with respect to P.*

Since the complement of a language accepted by a deterministic FCA can also be accepted by an FCA [I78], we also have:

Theorem 4. *If I and P (the safe set) are two sets of configurations of $A \oplus M$, let $GOOD$ be the set of all configurations in I that can only reach configurations in P. If I can be accepted by an FCA and P can be accepted by a deterministic FCA, then we can decide whether $GOOD = I$. Hence, invariance is decidable with respect to P.*

We can show that forward reachability is computable.

Theorem 5. *Let I be a set of configurations accepted by an FCA. We can effectively construct an acceptor in $\mathcal{C}$ accepting $post^*(A \oplus M, I) =$ the set of all configurations reachable from configurations in I.*

Similarly, for backward reachability we have:

Theorem 6. *Let I be a set of configurations accepted by an FCA. We can effectively construct an acceptor in $\mathcal{C}$ accepting $pre^*(A \oplus M, I) =$ the set of all configurations that can reach configurations in I.*

We can equip $A \oplus M$ with a one-way input tape. In order to do this, we can simply change the format of the transition edge of A by a 5-tuple $\langle s, \lambda, l, s', a \rangle$ in E, where a denotes an input symbol or ϵ (the null string). The meaning of this edge is like before, but now A can read a symbol or a null string at each transition. We also define a subset of the states of A as accepting states. Then $A \oplus M$ becomes an acceptor. Note that A and M will now start on some prescribed initial configurations (e.g., A is initialized to its start state with all clocks zero, M is initialized to its start state with all counters zero and the other data structures properly initialized). We can prove:

Theorem 7. *It is decidable to determine, given an acceptor $A \oplus M$, whether $A \oplus M$ accepts the empty set.*

One can extend the $A \oplus M$ acceptor to have multiple input tapes. Then like Theorem 1, we have:

Corollary 1. *It is decidable to determine, given a multitape acceptor $A \oplus M$, whether $A \oplus M$ accepts the empty set.*

We can also equip the multitape $A \oplus M$ acceptor with one-way output tapes. But, clearly, these output tapes can also be viewed as input tapes (since writing can be simulated by reading). Hence, the analysis of a multi-input-tape multi-output-tape $A \oplus M$ reduces to the analysis of multi-input-tape $A \oplus M$.

3 Examples of $\mathcal{C}$

We sketch the proof of Theorem 2 for the class $\mathcal{C}$, where each machine is a nondeterministic machine with a pushdown stack and finitely many reversal-bounded counters. Call a machine in this class a PCM, and PCA when it has an input tape (i.e., it is an acceptor). It is known that the emptiness problem for PCAs is decidable [I78]. Let A be a discrete timed automaton and M be a PCM. We describe precisely how $A \oplus M$ operates.

A configuration of the timed automaton A is of the form (s, U), where s is the state and U is the set of clock values. Now machine M has states, pushdown stack, and reversal-bounded counters. A move of M is defined by a transition function δ. If $\delta(q, Z, s_1, ..., s_k) = \{t_1, ..., t_m\}$, then

- q is the state, Z is the topmost symbol, and s_i is the status of counter i (i.e., zero or non-zero).
- $t_1, ..., t_m$ are the choices of moves (note that M is nondeterministic). Each t_i is of the form $(p, w, d_1, ..., d_k)$, which means pop Z and push string w (which is possibly empty) onto the stack, increment counter i by d_i (1, 0, or -1), and enter state p.

A configuration of M can be represented by a tuple of the form (q, V, w), where q is a state, V is the set of values of the counters, and w is the content of the stack with the rightmost symbol the top of the stack.

A transition of the combined machine $A \oplus M$ is now a tuple $\langle s, \lambda, l, s', ENTER(M, R)\rangle$, where $\langle s, \lambda, l, s'\rangle$ is as in a timed automaton. The combined transition is now carried out in two stages. Like before, A (the timed automaton component of the combined machine) makes the transition based on $\langle s, \lambda, l, s'\rangle$. It then transfers control to machine M by executing the command $ENTER(M, R)$, where R is a one-step transition rule: $R(s, \lambda, l, s', q, Z, s_1, ..., s_k) = \{t_1, ..., t_m\}$. Note that outcome this transition (i.e., the right side of the rule) not only depends on $\langle s, \lambda, l, s'\rangle$, but also on the current state, status of the counters, and the topmost symbol of the stack). This R is then followed by a sequence of transitions by M (using the transition function δ). Thus the use of $ENTER(M, R)$ allows the combined machine to update the configuration of M through a sequence of M's transitions. After some amount of computation, M returns control to A by entering a special state or command *RETURN*. When this happens, A will now be in state s'. Thus the computation of $A \oplus M$ is like in a timed automaton, except that between each transition of A, the system calls M to do some computation.

A configuration of the system is a tuple of the form $\alpha = (s, U, q, V, w)$. Thus, a configuration is one after the execution of a (possibly empty) sequence of *(ENTER, RETURN)* commands. Note that a configuration can be represented as a string where the clock values U and counter values V are represented in unary and the components of the tuple separated by markers.

As defined earlier, the binary reachability is $Reach(A \oplus M)$ = the set of all pairs of configurations (α, β), where α can reach β. We will show that $Reach(A \oplus M)$ can be accepted by a 2-tape PCA. Note that the input to the acceptor is a pair of strings (α, β), where α (β) is on the the first (second) tape.

First we note that we can view the clocks in a discrete timed automaton A as counters, which we shall also refer to as clock-counters. In a reversal-bounded multicounter machine, only *standard tests* (comparing a counter against 0) and *standard assignments* (increment or decrement a counter by 1, or simply no-change) are allowed. But clock-counters in A do not have standard tests nor standard assignments. The reasons are as follows. A clock constraint allows comparison between two clocks like $x_2 - x_1 > 7$. Note that using only standard tests we cannot directly compare the difference of two clock-counter values against an integer like 7 by computing $x_2 - x_1$ in another counter, since each time this computation is done, it will cause at least a counter reversal, and the number of such tests during a computation can be unbounded. The clock progress $x := x + 1$ is standard, but the clock reset $x := 0$ is not. Since there is no bound on the number of clock resets, clock-counters may not be reversal-bounded (each reset causes a counter reversal).

We first prove an intermediate result. Define a semi-PCA as a PCA which, in addition to a stack and reversal-bounded counters, has clock-counters that use nonstandard tests and assignments as described in the preceding paragraph.

Lemma 1. *We can effectively construct, given a discrete timed automaton A and and PCM M, a 2-tape semi-PCA B accepting $Reach(A \oplus M)$.*

Proof. We describe the construction of the 2-tape semi-PCA B. Given a pair of configurations (α, β) on its two input tapes, B first copies α into its counters and stack (these include the clock-counters). Then B simulates the ("alternating" mode of) computation of $A \oplus M$ starting from configuration α as described above. It is clear that B can do this. After some time, B guesses that it has reached the configuration β. It then checks that the values of the counters and stack match those on the second input tape. B accepts if the check succeeds. However, there is a slight complication because the pushdown stack content is in "reverse". If the stack content on the second tape is written in reversed, there is no problem. One can get around this difficulty if the comparison of the stack content with the second tape is done *during* the simulation instead of waiting until the end of the simulation. This involves guessing, for each position of the stack, the last time M rewrites this position, i.e., that the symbol would not be rewritten further in reaching configuration β. We omit the details. ∎

The next lemma converts the 2-tape semi PCA to a 2-tape PCA. The proof uses a technique in [DIBKS00] (see also [IDS00]).

Lemma 2. *We can effectively construct from the 2-tape semi-PCA B, a 2-tape PCA C equivalent to B.*

Proof. The 2-tape PCA C operates like B, but the simulation of $A \oplus M$ differs in the way A is simulated. Let A have clock-counters $x_1, ..., x_k$. Let m be one plus the maximal absolute value of all the integer constants that appear in the tests (i.e., the clock constraints on the edges of A in the form of Boolean combinations of $x_i \# c$, $x_i - x_j \# c$ with c an integer). Denote the finite set $\{-m, \cdots, 0, \cdots, m\}$ by $[m]$. Define two finite tables with entries a_{ij} and b_i for $1 \leqslant i, j \leqslant k$. Each entry can be regarded as a finite state variable with states in $[m]$. Intuitively, a_{ij} is used to record the difference between two clock values of x_i and x_j, and b_i is used to record the clock value of x_i. During the computation of A, when the difference $x_i - x_j$ (or the value x_i) goes above m or below $-m$, a_{ij} (or b_i) stays the same as m or $-m$.

C simulates A exactly except that it uses $a_{ij} \# c$ for the test $x_i - x_j \# c$ and $b_i \# c$ for the test $x_i \# c$, with $-m < c < m$. One can prove (by induction) that doing this is valid: Each time after C updates the entries by executing a transition, $x_i - x_j \# c$ iff $a_{ij} \# c$, and $x_i \# c$ iff $b_i \# c$, for all $1 \leqslant i, j \leqslant k$ and for each integer $c \in [m-1]$. Th details for setting the initial values of the entries of a_{ij} an b_i) and updating them are given in the full paper.

Thus clock-counter comparisons are replaced by finite table look-up and, therefore, nonstandard tests are not present in C. Finally, we show how nonstandard assignments of the form $x_i := 0$ (clock resets) in machine C can be avoided.

Clearly after eliminating the clock comparisons, the clock-counters in C do not participate in any tests except:

- At the beginning of the simulation when the initial values of the x_i's are used to compute the initial values of the a_{ij}'s and the b_i's as described above;
- At the end of the simulation when the final values of the x_i's are compared with the second input tape to check whether they match those in β.

Thus, for each x_i, during the simulation of A but before the last reset of x_i, the actual value of x_i is irrelevant. We describe how to construct a 2-tape PCA D from C such that in the simulation of A, no nonstandard assignment is used. For each clock x_i in A, there are two cases. The first case is when x_i will not be reset during the entire simulation of C. The second case is when x_i will be reset. D guesses the case for each x_i. For the first case, x_i is already reversal-bounded, since the nonstandard assignment $x_i := 0$ is not used. For the second case, D first decrements x_i to 0. Then D simulates C. Whenever a clock progress $x_i := x_i + 1$ or a clock reset $x_i := 0$ is being executed by A, D keeps x_i as 0. But, at some point when a clock reset $x_i := 0$ is being executed by A, D guesses that this is the last clock reset for x_i. After this point, D faithfully simulates a clock progress $x_i := x_i + 1$ executed by A, and a later execution of a clock reset $x_i := 0$ in A will cause D to abort abnormally (since the guess of the last reset of x_i was wrong). Thus D uses only standard assignments $x_i := x_i + 1, x_i := x_i$, and $x_i := x_i - 1$ initially to bring x_i to 0 (for the second case). ∎

From the above lemmas, we have:

Theorem 8. *We can effectively construct, given a discrete timed automaton A and and a PCM M, a 2-tape PCA accepting Reach*$(A \oplus M)$.

One can generalize Theorem 8. Extend a PCA acceptor by allowing the machine to have multiple pushdown stacks. Thus the machine will have multiple reversal-bounded counters and multiple stacks (ordered by name, say $S_1, ..., S_m$). The operation of the machine is restricted in that it can only read the topmost symbol of the *first* nonempty stack. Thus a move of the machine would depend only on the current state, the input symbol (or ϵ), the status of each counter (zero or nonzero), and the topmost symbol of the first stack, say S_i, that is not empty (initially, all stacks are set to some starting top symbol). The action taken in the move consists of the input being consumed, each counter being updated (+1, -1, 0), the topmost symbol of S_i being popped and a string (possibly empty) being pushed onto each stack, and the next state being entered. This acceptor, call it MPCA, was studied in [D00] as a generalization of a PCA [I78] and a generalization of a multipushdown acceptor [CBCC96]. Thus an MPCA with only one stack reduces to a PCA.

By combining the techniques in [I78] and [CBCC96], it was shown in [D00] that the emptiness problem for MPCAs is decidable. An MPCA without an input tape will be called an MPCM. By a construction similar to that of Theorem 8, we can prove the next result. Note that checking that the contents of the stacks at the end of the simulation are the same as the stack words in the target configuration does not require the latter to be in reverse (or need special handling), since we can first reverse the stack contents by using another set of pushdown stacks and then check that they match the stack words in the target configuration.

Theorem 9. *We can effectively construct, given a discrete timed automaton A and an MPCM M, a 2-tape MPCA accepting Reach*$(A \oplus M)$.

Other examples of classes $\mathcal{C}$ that can be shown to have a decidable emptiness problem are given below. Thus, the results in Section 2 apply.

1. Nondeterministic machines with reversal-bounded counters and a two-way read/write worktape that is restricted in that the number of times the head crosses the boundary between any two adjacent cells of the worktape is bounded by a constant, independent of the computation (thus, the worktape is finite-crossing). There is no bound on how long the head can remain on a cell [IBS00].
2. Nondeterministic machines with reversal-bounded counters and a queue that is restricted in that the number of alternations between non-deletion phase and non-insertion phase is bounded by a constant [IBS00]. A non-deletion (non-insertion) phase is a period consisting of insertions (deletions) and no-changes, i.e., the queue is idle. Without the restriction emptiness is undecidable since it is known that a finite-state machine with an unrestricted queue can simulate a Turing machine.

Finally, as mentioned in the paragraph preceding Theorem 7, we can provide the machine $A \oplus M$ with an input tape. The language accepted by such an acceptor can be shown to be accepted by an acceptor M' which belongs to the same class as M (the simulation is similar to the one described in Lemmas 1 and 2). Thus, Theorem 7 follows.

4 Applications

In this section we exhibit some properties of timed automata that can be verified using the results above.

Example 1. (Real-time) pushdown timed systems with "observation" counters were studied in [BER95]. The purpose of these counters is to record information about the evolution of the system and to reason about certain properties (e.g., number of occurrences of certain events in some computation). The counters do not participate in the dynamic of the system, i.e., they are never tested by the system. A transition edge specifies for each observation counter an integral value (positive, negative, zero) to be added to the counter. Of interest are the values of the counters when the system reaches a specified configuration. It was shown in [BER95] that region reachability is decidable for these systems.

Clearly, for the discrete case, such a system can be simulated by the machine $A \oplus M$ described in the previous section. We associate in M two counters for each observation counter: one counter keeps track of the positive increases and the other counter keeps track of the negative increases. When the target configuration is reached, the difference can be computed in one of the counters. Note that the sign of the difference can be specified in another counter, which is set to 0 for negative and 1 for positive. Thus, from Theorems 2–6, (binary, forward, backward) reachability, safety, and invariance are solvable for these systems.

Example 2. Let A be a discrete timed automaton and M be a nondeterministic pushdown machine with reversal-bounded counters. For a given computation of $A \oplus M$, let r_i be the number of times clock x_i resets. Suppose we are interested in computations in which the r_i's satisfy a Presburger formula f, i.e., we are interested in (α, β) in $Reach(A \oplus M)$ such that α can reach β in a computation in which the clock resets satisfy f. It is known that a set of k-tuples is definable by a Presburger formula f if and only if it is definable by a reversal-bounded multicounter machine [I78]. (Thus, a machine M_f with no input tape but with reversal-bounded counters can be effectively constructed from f such that when the values of the first k counters are set to the k-tuple and all the other counters initially zero, M_f enters an accepting state if and only if the k-tuple satisfies f. In fact, M_f can be made deterministic [I78].) It follows that we can construct a 2-tape pushdown acceptor with reversal-bounded counters M' accepting the set Q of pairs of configurations (α, β) in $Reach(A \oplus M)$ such that α can reach β in a computation in which the clock resets satisfy f. One can also put other

constraints, like introducing a parameter t_i for each clock i, and consider computations where the first time i resets to zero is before (or after) time t_i. We can construct a 3-tape acceptor M'' from M' accepting $Q(t_1, ..., t_k)$. M'' first reads the parameters t_i's (which are given on the third input tape) and then simulates M', checking that the constraint on the first time clock i resets is satisfied. Note that if M has no pushdown stack, then Q and $Q(t_1, ..., t_k)$ are Presburger.

Example 3. As another example, suppose we are interested in the set S of pairs of configurations (α, β) of a discrete timed automaton A such that there is a computation path (i.e., sequence of states) from α to β that satisfies a property that can be verified by an acceptor in a class C. If C has a decidable emptiness problem, then S is effectively computable. For example, suppose that the property is for the path to contain three non-overlapping subpaths (i.e., segments of computation) which go through the same sequence of states, and the length of the subpath is no less than 1/5 of the length of the entire path. Thus if p is the computation path, there exist subpaths $p_1, ..., p_7$ (some may be null) such that $p = p_1p_2p_3p_4p_5p_6p_7$, where p_2, p_4, and p_6 go through the same sequence of states, and length of p_2 = length of p_4 = length of p_6 is no less than 1/5 of the length of p. We can check this property by incorporating a finite-crossing read-write tape to the machine (actually, the head need only make 5 crossings on the read-write tape).

Example 4. We can equip $A \oplus M$ with one-way write-only tapes which the machine can use to record certain information about the computation of the system (and perhaps even requiring that the strings appearing in these tapes satisfy some properties). From Corollary 1, such systems can effectively be analyzed.

5 Reachability in Parallel Discrete Timed Automata

The technique of using the reversal-bounded counters to record and compare various integers (like the running times of the machines) in the proofs in Section 3 can be used to decide some reachability questions concerning machines operating in parallel. We give two examples below.

Let A_1, A_2 be discrete timed automata and M_1, M_2 be PCMs. Recall from Section 3 that a configuration of $A_i \oplus M_i$ is a 5-tuple $\alpha_i = (s_i, U_i, q_i, V_i, w_i)$. Suppose we are given a pair of configurations (α_1, β_1) of $A_1 \oplus M_1$ and a pair of configurations (α_2, β_2) of $A_2 \oplus M_2$, and we want to know if $A_i \oplus M_i$ when started in configuration α_i can reach configuration β_i at some time t_i, with t_1 and t_2 satisfying a given linear relation $L(t_1, t_2)$ definable by a Presburger formula. (Thus, e.g., if the linear relation is $t_1 = t_2$, then we want to determine if $A_1 \oplus M_1$ when started in configuration α_1 reaches β_1 at the same time that $A_2 \oplus M_2$ when started in α_2 reaches β_2.) This reachability question is decidable. The idea is the following. First note that we can incorporate a counter in M_i that records the running time t_i of $A_i \oplus M_i$. Let Z_i be a 2-tape PCA accepting

$R(A_i \oplus M_i)$. We construct a 4-tape PCA Z which, when given $\alpha_1, \beta_1, \alpha_2, \beta_2$ in its 4 tapes, first simulates the computation of Z_1 to check that α_1 can reach β_1, recording the running time t_1 (which is in configuration β_1) of $A_1 \oplus M_1$ in a counter. Z then simulates Z_2. Finally, Z checks that the running times t_1 and t_2 satisfy the given linear relation (which can be verified since Presburger formulas can be evaluated by nondeterministic reversal-bounded multicounter machines). Since the emptiness problem for PCAs is decidable, decidability of reachability follows.

We can allow the machines $A_1 \oplus M_1$ and $A_2 \oplus M_2$ to share a common input tape, i.e., each machine has a one-way read-only input head (see the paragraph preceding Theorem 7). A configuration α_i will now be a 7-tuple $\alpha_i = (s_i, U_i, q_i, V_i, w_i, h_i)$, h_i is the position of the input head on the common input x. One can show that if both $A_1 \oplus M_1$ and $A_2 \oplus M_2$ have a one-turn stack (or an unrestricted counter), then reachability is undecidable, even if they have no reversal-bounded counters and the linear relation is $t_1 = t_2$. However, if only one of $A_1 \oplus M_1$ and $A_2 \oplus M_2$ has an unrestricted pushdown stack, then reachability is decidable. The idea is to construct a 5-tape PCA which, when given $\alpha_1, \beta_1, \alpha_2, \beta_2, x$, simulates M_1 and M_2 in parallel on the input x, recording their running times and then check that the linear relation is satisfied.

Note that the above results generalize to any number, k, of machines $A_i \oplus M_i$ $(i = 1, ..., k)$ operating in parallel.

6 Conclusions

We showed that a discrete timed automaton augmented with a machine with reversal-bounded counters and possibly other data structures from a class $\mathcal{C}$ of machines can be effectively analyzed with respect to reachability, safety, and other properties if $\mathcal{C}$ has a decidable emptiness problem. We gave examples of such $\mathcal{C}$'s and examples of new properties of discrete timed automata that can be verified. We also showed that reachability in parallel machines can be effectively decided. It would be interesting to look for other classes of $\mathcal{C}$'s with decidable emptiness problem.

References

[AD94] R. Alur and D. Dill. Automata for modeling real-time systems. *Theoretical Computer Science*, 126(2), pp. 83-236, 1994.

[BER95] A. Bouajjani, R. Echahed, and R. Robbana. On the Automatic Verification of Systems with Continuous Variables and Unbounded Discrete Data Structures. In *Hybrid Systems II*, LNCS 999, 1995.

[CBCC96] A. Cherubini, L. Breveglieri, C. Citrini, and S. C. Reghizzi. Multiple pushdown languages and grammars. *Int. J. of Foundations of Computer Science*, pp. 253-291, 1996.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. *Proc. 10th Int. Conf. on Computer Aided Verification*, pp. 268-279, 1998.

[CJ99] H. Comon and Y. Jurski. Timed automata and the theory of real numbers. *Proc. 10th Int. Conf. on Concurrency Theory*, pp. 242-257, 1999.

[D00] Z. Dang. Verification and Debugging of Infinite State Real-time Systems. *Ph.D. Thesis*. University of California, Santa Barbara, 2000.

[DIBKS00] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. Binary reachability analysis of discrete pushdown timed automata. *Int. Conf. on Computer Aided Verification*, 2000.

[I78] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. of the Association for Computing Machinery*, 25, pp. 116-133, 1978.

[IBS00] O. H. Ibarra, T. Bultan, and J. Su. Reachability analysis for some models of infinite-state transition systems. *Proc. 10th Int. Conf. on Concurrency Theory*, 2000.

[IDS00] O. H. Ibarra, Z. Dang, and P. San Pietro. Queue-Connected Discrete Timed Automata. In preparation.

[M61] M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. *Ann. of Math.*, 74, pp. 437-455, 1961.

Factorization of Ambiguous Finite-State Transducers

André Kempe

Xerox Research Centre Europe – Grenoble Laboratory
6 chemin de Maupertuis – 38240 Meylan – France
andre.kempe@xrce.xerox.com – http://www.xrce.xerox.com/research/mltt

Abstract. This article describes an algorithm for factorizing a finitely ambiguous finite-state transducer (FST) into two FSTs, T_1 and T_2, such that T_1 is functional and T_2 retains the ambiguity of the original FST. The application of T_2 to the output of T_1 never leads to a state that does not provide a transition for the next input symbol, and always terminates in a final state. In other words, T_2 contains no "failing paths" whereas T_1 in general does. Since T_1 is functional, it can be factorized into a left-sequential and a right-sequential FST that jointly constitute a bimachine. The described factorization can accelerate the processing of input because no failing paths are ever followed.

1 Introduction

An ambiguous finite-state transducer (FST) returns for every accepted input string one or more output strings by following different alternative paths from the initial state to a final state. In addition, there may be a number of other paths that are followed from the initial state up to a certain point where they fail. Following these latter paths is necessary but at the same time inefficient.

We present an algorithm for factorizing (decomposing) a finitely ambiguous[1] FST into two FSTs, T_1 and T_2, such that T_1 is functional and T_2 retains the ambiguity of the original FST. We call T_2 *fail-safe*, meaning that its application to the output of T_1 never leads to a state that does not provide a transition for the next input symbol, and always terminates in a final state.

Because T_1 is functional, it can be further factorized into a left-sequential[2] and a right-sequential FST, T_{11} and T_{12}, that jointly constitute a *bimachine* as introduced by Schützenberger [9], using an existing factorization algorithm [3,2,7]. The resulting three FSTs, T_{11}, T_{12}, and T_2, are used in a cascade that simulates composition.

[1] Since infinite ambiguity, described by ε-loops, usually does not occur in practical applications, the limitation of the algorithm to finitely ambiguous FSTs does not constitute an obstacle in practice.

[2] The terms *left-deterministic*, *left-sequential*, etc. actually mean *left-to-right-deterministic*, *left-to-right-sequential*, etc. Similarly, *right-deterministic* means *right-to-left-deterministic* etc.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 170–181, 2001.

Another method for factorizing an ambiguous FST can be derived from a construction on automata by Schützenberger [10], clearer described by Sakarovitch [8, Sec. 3] in the framework of the so-called *covering of automata*.[3] This factorization would yield a different result than the one described below.

Factorization of FSTs can be useful for many practical applications, e.g. in Natural Language Processing where FSTs are used for many basic steps [4,6]. It can accelerate the processing of input because no time is spent on failing paths, and allows analyzing and manipulating separately the different parts of an FST (or of the described relation).

1.1 Conventions

Every FST has one initial state, labeled with number 0, and one or more final states marked by double circles. An arc with n labels designates a set of n arcs with one label each that all have the same source and destination. In a symbol pair occurring as an arc label, the first symbol is the input and the second the output symbol. For example, in the pair `a:b`, `a` is the input and `b` the output symbol. Unpaired symbols represent identity pairs. For example, `a` means `a:a`.

2 Basic Idea

An FST can contain a number of failing paths for a given input string. The FST in Example 1 (Fig.1) contains for the input string `cabca` two successful paths, formed by the ordered arc sets ⌈*101, 104, 108, 112, 115*⌉ and ⌈*101, 104, 109, 113, 115*⌉ respectively, and three failing paths, ⌈*100, 102, 105*⌉, ⌈*100, 102, 106*⌉, and ⌈*100, 103, 107*⌉. For the string `caba` it has no successful and five failing paths, ⌈*100, 102, 105*⌉, ⌈*100, 102, 106*⌉, ⌈*100, 103, 107*⌉, ⌈*101, 104, 108*⌉, and ⌈*101, 104, 109*⌉. Following all failing paths is inevitable but inefficient.

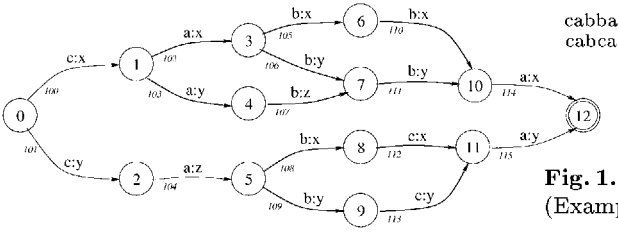


Fig. 1. Ambiguous ε-free FST T (Example 1)

Any ambiguous ε-free FST T can be factorized into two FSTs, T_1 and T_2, such that T_1 is unambiguous and T_2 is fail-safe wrt. the output of T_1 (Fig. 2).

[3] Many thanks to Jacques Sakarovitch (CNRS and ENST, Paris) for pointing me to this work [10,8] and for explaining how Schützenberger's construction can be made the principle step in the factorization of ambiguous FSTs.

Because of its structure, T_2 is called a *flower transducer*. Informally spoken, the factorization algorithm collapses a set of alternative sub-paths of T into one single sub-path in T_1, and expands it again in T_2. When applied to an input string, T_1 and T_2 operate as a cascade: T_1 maps the input string to (at most) one intermediate string, and T_2 maps that string to a set of alternative output strings.

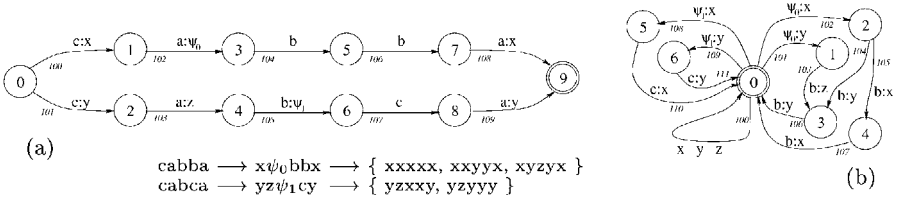


Fig. 2. Factorization of T into (a) a functional T_1 and (b) an ambiguous fail-safe flower transducer T_2 (Example 1)

The FST in Example 1 contains two *ambiguity fields* (Fig. 1). An ambiguity field is a maximal set of alternative subpaths that all accept the same substring in the same position of the same input strings. The first ambiguity field in Example 1 spans from state 1 to 10, and maps the substring `abb` of the input string `cabba` to the set of alternative output substrings {`xxx, xyy, yzy`}. In T_1 this ambiguity field is collapsed into a single subpath ranging from state 1 to 7 that maps the substring `abb` to ψ_0`bb` (Fig.2a). T_2 maps this intermediate substring to the set of output substrings {`xxx`, `xyy`, `yzy`} by following the alternative subpaths ⌈*102, 105, 107*⌉, ⌈*102, 104, 106*⌉, and ⌈*101, 103, 106*⌉ respectively (Fig. 2b). The second ambiguity field of Example 1 spans from state 5 to 11, and maps the substring `bc` of the input string `cabca` to the set of output substrings {`xx`, `yy`} (Fig. 1). In T_1 this ambiguity field is collapsed into a single subpath ranging from state 4 to 8 that maps `bc` to ψ_1`c` (Fig. 2a). T_2 maps the latter substring to the output substrings {`xx`, `yy`} by following the subpaths ⌈*108, 110*⌉ and ⌈*109, 111*⌉ respectively (Fig. 2b).

Note that in T_1 a diacritic is used only on the first arc of a collapsed ambiguity field, and that the other arcs of the ambiguity field (usually) simply map an input symbol to itself. All symbols that are accepted outside an ambiguity field, are mapped in T_1 to their final output which is then mapped to itself in T_2, by an arc that loops on the initial state (Fig. 2). In the current example this loop consists of the arc *100* that is actually a set of three looping arcs with one symbol each (Fig. 2b).

T_1, which is functional but not sequential, can be further factorized into a left-sequential and a right-sequential FST, T_{11} and T_{12}, that jointly constitute a bimachine. The three FSTs, T_{11}, T_{12}, and T_2, together represent a factorization of T. The factorization from Example 1 is shown in Figure 3. When applied to an input string, the three FSTs operate as a cascade: T_{11} maps, e.g., the

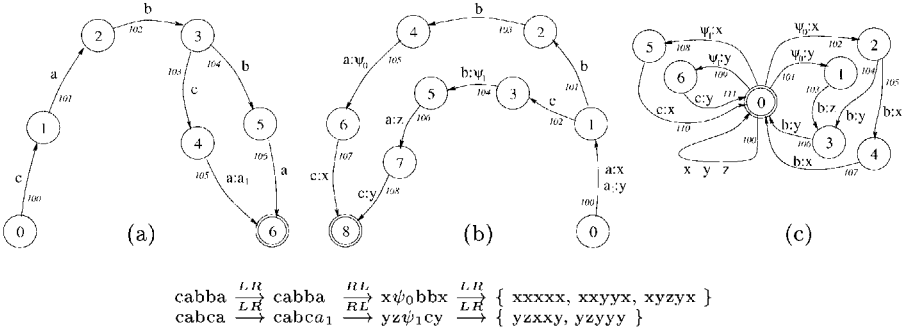


Fig. 3. Factorization of T into (a) a left-sequential T_{11}, (b) a right-sequential fail-safe T_{12}, and (c) an ambiguous fail-safe T_2 (Example 1)

input string `cabca`, deterministically from left to right, to the intermediate string `cabc`a_1 (Fig. 3a). T_{12} maps then this string, deterministically from right to left, to `yz`ψ_1`cy` (Fig. 3b). Finally, T_2 maps that string, from left to right, to the set of alternative output strings {`yzxxy`, `yzyyy`} (Fig. 3c). In such a cascade, T_{11} and T_{12} are sequential, and T_{12} and T_2 are fail-safe wrt. the output of their predecessor. Input strings that are not accepted, fail in the first FST, T_{11}, on one single path, and require no further attention.

3 Factorization Algorithm

3.1 Starting Point

The factorization of the ambiguous ε-free FST in Example 2 (Fig. 4) requires identifying maximal sets of *alternative arcs* that must be collapsed in T_1 and expanded again in T_2. Two arcs are alternative wrt. each other if they are situated at the same position on two *alternative paths* that accept the same input string. This means the two arcs must have the same input symbol and equal sets of *input prefixes* and *input suffixes*. The two arcs *105* and *106* in Example 2 constitute such a maximal set of alternative arcs (Fig. 4). Both arcs have the input symbol `b`, the input prefix set {`a`*`ab`}, and the input suffix set {`ca`, `cb`, `cc`}. Two arcs are not alternative wrt. each other and must not be collapsed if they have either different input symbols, or no prefixes or no suffixes in common.

In general, an FST can contain arcs where none of these two premises is true. In Example 2 the arcs *103* and *104* have identical input symbols, `b`, and equal input prefix sets, {`a`*`a`}, but their input suffix sets, {ε, `bca`, `bcb`, `bcc`} and {`bca`, `bcb`, `bcc`} respectively, are neither equal nor disjoint (Fig. 4). These two arcs are only *partially alternative* which means they must be collapsed and not collapsed at the same time. To resolve this dilemma, the FST must be transformed (pre-processed) such that the sets of input prefixes and input suffixes of all arcs become either equal or disjoint, without changing the relation described by the FST.

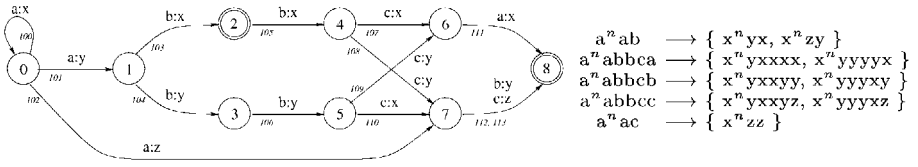


Fig. 4. Ambiguous ε-free FST T (Example 2)

3.2 Pre-processing

The first step of the pre-processing consists of concatenating the FST on both sides with boundary symbols, #, and minimizing the result by means of standard algorithms [1] (Fig. 5). This operation "transfers" the properties of initiality and finality from states to special arcs. Therefore, these properties will not require any attention in some subsequent operations. The result of the first pre-processing step will be referred to as *minimal FST* T^m.

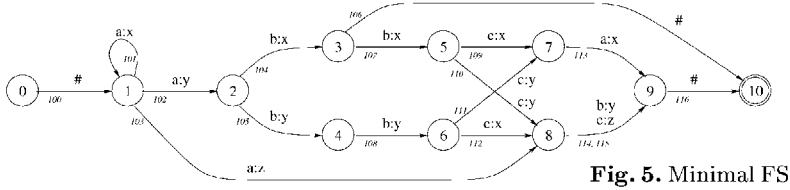


Fig. 5. Minimal FST T^m with boundaries (Example 2)

The second step of the pre-processing consists of a *left-unfolding* of T^m, which means that every state q^m of T^m is split into a set of children states q_i. Each prefix of q^m is inherited by only one q_i, and each suffix by all of them. Consequently, different q_i of the same q^m have disjoint prefix sets and equal suffix sets (Fig. 6).

The operation is based on the *left-deterministic input automaton* A^L of T^m which is obtained by extracting the input side from T^m, and determinizing it from left to right (Fig. 6a). Every state of A^L corresponds to a set of states of T^m, and is assigned the set of corresponding state numbers (Fig. 5, 6a). Every state of T^m is copied to the *left-unfolded FST* T^L (Fig. 6b) as many times as it occurs in different state sets of A^L. (The copying of the arcs is described later.) For example, state 8 of T^m occurs in the states sets of both state 2 and 5 of A^L, and is therefore copied twice to T^L, where the two copies have the state numbers 9 and 10.

Every state q of T^L corresponds to one state q^m of T^m and to one state q^L of A^L. In the left-unfolded T^L of Example 2, every state is labeled with a triple of state numbers $\langle q, q^m, q^L \rangle$ (Fig. 6b). For example, states 9 and 10 are labeled with the triples $\langle 9, 8, 5 \rangle$ and $\langle 10, 8, 2 \rangle$ respectively which means that they are

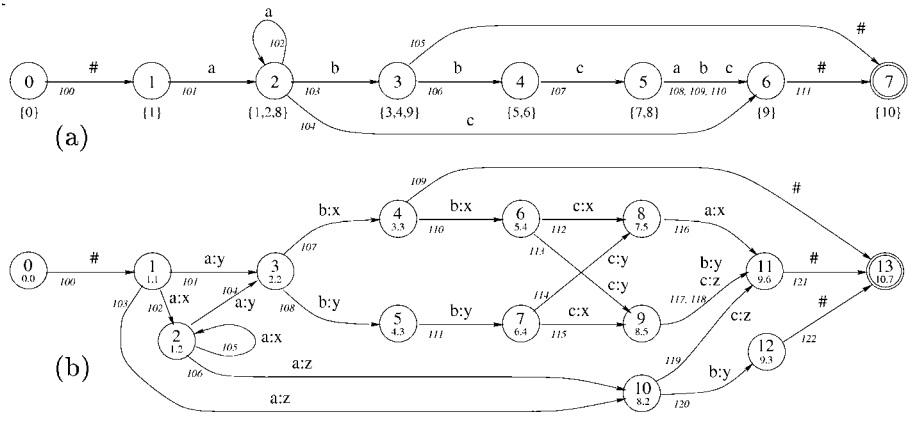


Fig. 6. (a) Left-deterministic input automaton A^L, built from T^m, and (b) left-unfolded FST T^L (Example 2)

both copies of state 8 ($=q^m$) of T^m but correspond to different states of A^L, namely to the states 5 and 2 ($=q^L$) respectively.

Every state q of T^L inherits the full set of outgoing arcs of the corresponding state q^m of T^m. For example, the set of outgoing arcs {*101, 102, 103*} of state 1 ($=q^m$) of T^m is inherited by both state 1 and 2 ($=q^L$) of T^L where it becomes {*102, 101, 103*} and {105, 104, 106} respectively (Fig. 5, 6b). The destination of every arc of T^L is determined by the destination states q^m and q^L of the corresponding arcs in T^m and A^L. For example, the arc *102* of T^L must point to state 2, labeled with $\langle 2, 1, 2\rangle$, because this (and only this) state corresponds to both the destination $q^m = 1$ of the corresponding arc *101* in T^m and the destination $q^L = 2$ of the corresponding arc *101* in A^L.

The left-unfolded T^L describes the same relation as T^m. Minimizing T^L would generate T^m.

The third step of the pre-processing consists of a *right-unfolding* of the previously left-unfolded T^L, which means that every state q of T^L is split into a set of children states q_i. Each prefix of q is inherited by all q_i, and each suffix by only one of them. Consequently, different q_i of the same q have equal prefix sets and disjoint suffix sets (Fig. 7).

The operation is based on the *right-deterministic input automaton* A^R of the previously left-unfolded T^L, and is performed exactly as the second step, except that T^L is reversed before the operation, and reversed back afterwards. The reversal consists of making the initial state final and the only final state initial, and changing the direction of all arcs, without minimization or determinization that would change the structure of the FST.

Every state q of the fully (i.e. left- and right-) unfolded FST $T^{L,R}$ (Fig. 7b) corresponds to one state q^m of T^m, to one state q^L of A^L, and to one state q^R of A^R. In the fully unfolded $T^{L,R}$ of Example 2, every state is labeled with a quadruple of state numbers $\langle q, q^m, q^L, q^R\rangle$ (Fig. 7b). For example, the states 11, 12, 13,

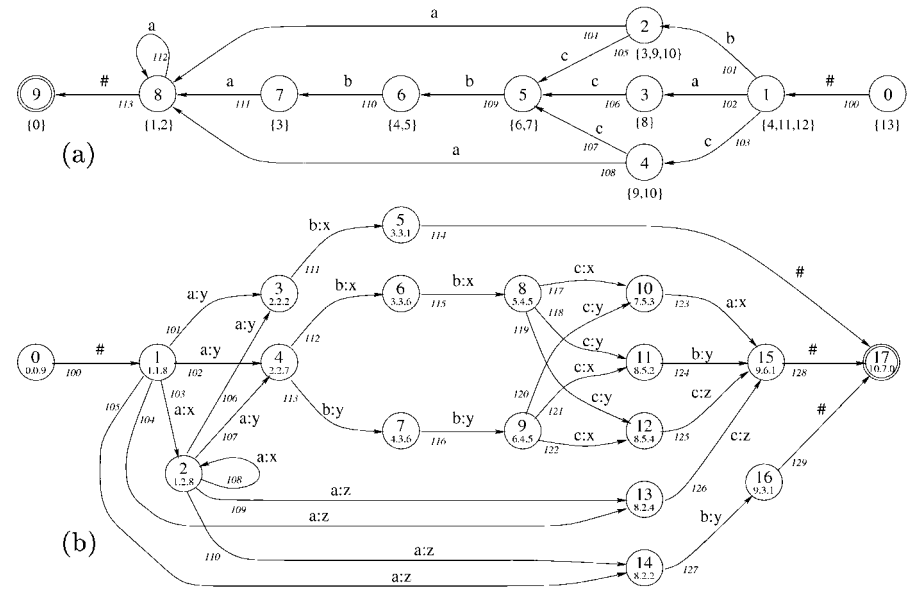


Fig. 7. (a) Right-deterministic input automaton A^R, built from the left-unfolded FST T^L, and (b) fully (i.e. left- and right-) unfolded FST $T^{L,R}$ (Example 2)

and 14 are labeled with the quadruples $\langle 11, 8, 5, 2\rangle$, $\langle 12, 8, 5, 4\rangle$, $\langle 13, 8, 2, 4\rangle$, and $\langle 14, 8, 2, 2\rangle$ which means that they are all copies of state 8 ($= q^m$) of T^m but corresponds to different states of A^L and A^R.

Every state q of $T^{L,R}$ has the same input prefix set $\mathcal{P}^{in}(q)$ as the corresponding state q^L of A^L and the same input suffix set $\mathcal{S}^{in}(q)$ as the corresponding state q^R of A^R:

$$\forall q \in Q \ : \qquad \mathcal{P}^{in}(q) = \mathcal{P}^{in}(q^L) \tag{1}$$

$$\mathcal{S}^{in}(q) = \mathcal{S}^{in}(q^R) \tag{2}$$

Consequently, two states, q_i and q_j, of $T^{L,R}$ have equal input prefix sets iff they correspond to the same state q^L, and equal input suffix sets iff they correspond to the same state q^R:

$$\forall q_i, q_j \in Q \ : \qquad \mathcal{P}^{in}(q_i) = \mathcal{P}^{in}(q_j) \iff q_i^L = q_j^L \tag{3}$$

$$\mathcal{S}^{in}(q_i) = \mathcal{S}^{in}(q_j) \iff q_i^R = q_j^R \tag{4}$$

The input prefix and suffix sets of the states of $T^{L,R}$ are either equal or disjoint. Partial overlaps cannot occur.

Equivalent states of $T^{L,R}$ are different copies of the same state q^m of T^m. This means, two states, q_i and q_j, are equivalent iff they correspond to the same state q^m of T^m:

$$q_i \ \equiv \ q_j : \iff \ q_i^m = q_j^m \tag{5}$$

Every arc a of the fully unfolded $T^{L,R}$ can be described by a quadruple:

$$a = \langle s, d, \sigma^{in}, \sigma^{out} \rangle \qquad \text{with } a \in A;\ s, d \in Q;\ \sigma^{in} \in \Sigma^{in};\ \sigma^{out} \in \Sigma^{out} \tag{6}$$

where s and d are the source and destination state, and σ^{in} and σ^{out} the input and output symbol of the arc a respectively. For example, the arc *102* of $T^{L,R}$ can be described by the quadruple $\langle 1, 4, \mathtt{a}, \mathtt{y} \rangle$ (Fig. 7b).

Alternative arcs describe alternative transductions of the same input symbol in the same position of the same input string. Two arc, a_i and a_j, are alternative wrt. each other iff they have the same input symbol and equal input prefix and suffix sets. The input prefix set of an arc is the input prefix set of its source state, and the input suffix set of an arc is the input suffix set of its destination state:

$$a_i \overset{\text{alt}}{\sim} a_j \ :\Longleftrightarrow\ (\sigma_i^{in} = \sigma_j^{in}) \ \wedge\ (\mathcal{P}^{in}(s_i) = \mathcal{P}^{in}(s_j)) \ \wedge\ (\mathcal{S}^{in}(d_i) = \mathcal{S}^{in}(d_j)) \tag{7}$$

Equivalent arcs are different copies of the same arc of T^m. Two arcs are equivalent iff they have the same input and output symbol, and equivalent source and destination states:

$$a_i \ \equiv\ a_j : \Longleftrightarrow\ (\sigma_i^{in} = \sigma_j^{in}) \ \wedge\ (\sigma_i^{out} = \sigma_j^{out}) \ \wedge\ (s_i \equiv s_j) \ \wedge\ (d_i \equiv d_j) \tag{8}$$

Two equivalent arcs are also alternative wrt. each other but not vice versa.

The fully unfolded $T^{L,R}$ describes the same relation as T^m (Fig. 5). Minimizing $T^{L,R}$ would generate T^m. The previous dilemma of collapsing partially alternative arcs does not occur in $T^{L,R}$ where arcs are never partially alternative wrt. each other.

3.3 Construction of Factors

After the pre-processing, preliminary factors, T_1' and T_2', are built (Fig. 8) : First, all states of the unfolded $T^{L,R}$ are copied (as they are) to both T_1' and T_2'. Then, all arcs of $T^{L,R}$ are grouped to disjoint maximal sets $\mathcal{A}$ of alternative arcs. For the current example (Fig. 7b), the sets are:

{*100*}, {*101, 105*}, {*102*}, {*103*}, {*104*}, {*106, 110*}, {*107*}, {*108*}, {*109*},
{*111, 127*}, {*112, 113*}, {*114, 129*}, {*115, 116*}, {*117, 120*}, {*118, 121*},
{*119, 122*}, {*123*}, {*124*}, {*125*}, {*126*}, {*128*}

Sets $\mathcal{A}$ of alternative arcs can have the following different locations wrt. *ambiguity fields* (cf. Sec. 2):

- Singleton sets (e.g., {*100*} or {*102*} in Fig. 7b) and sets where all arcs are equivalent wrt. each other (no example in Fig. 7b) do not describe an ambiguity. These arc sets are outside any ambiguity field.
- All other arc sets $\mathcal{A}$ describe an ambiguity (e.g., {*115, 116*}). They are inside an ambiguity field where three different (possibly co-occurring) locations can be distinguished:
  - $\mathcal{A}$ is at the beginning of an ambiguity field iff the source states of all arcs in $\mathcal{A}$ are equivalent (e.g., {*101, 105*} and {*112, 113*}) :

$$Begin(\mathcal{A}) \ :\Longleftrightarrow\ \forall a_i, a_j \in \mathcal{A} \ :\ s_i \equiv s_j \tag{9}$$

- $\mathcal{A}$ is at the end of an ambiguity field iff the destination states of all arcs in $\mathcal{A}$ are equivalent (e.g., $\{117, 120\}$ and $\{114, 129\}$) :

$$End(\mathcal{A}) \;:\Longleftrightarrow\; \forall a_i, a_j \in \mathcal{A} \;:\; d_i \equiv d_j \tag{10}$$

- $\mathcal{A}$ is at an *ambiguity fork*, i.e., at a position where two or more ambiguity fields with a common (overlapping) beginning separate from each other, iff there is an arc a_i in $\mathcal{A}$ and an arc a_k outside $\mathcal{A}$ so that both have the same input symbol and equivalent source states but disjoint input suffix sets. This means that the state q^m of T^m, that corresponds to the source states of both arcs, can be left via either arc, a_i or a_k, but one of the arcs is on a failing path, and therefore should not be taken (e.g., $\{117, 120\}$ and $\{118, 121\}$) :

$$Fork(\mathcal{A}) \;:\Longleftrightarrow\; \exists a_i \in \mathcal{A}, \exists a_k \notin \mathcal{A} \;:\; (\sigma_i^{in} = \sigma_k^{in}) \wedge (s_i \equiv s_k) \wedge (\mathcal{S}^{in}(d_i) \neq \mathcal{S}^{in}(d_k)) \tag{11}$$

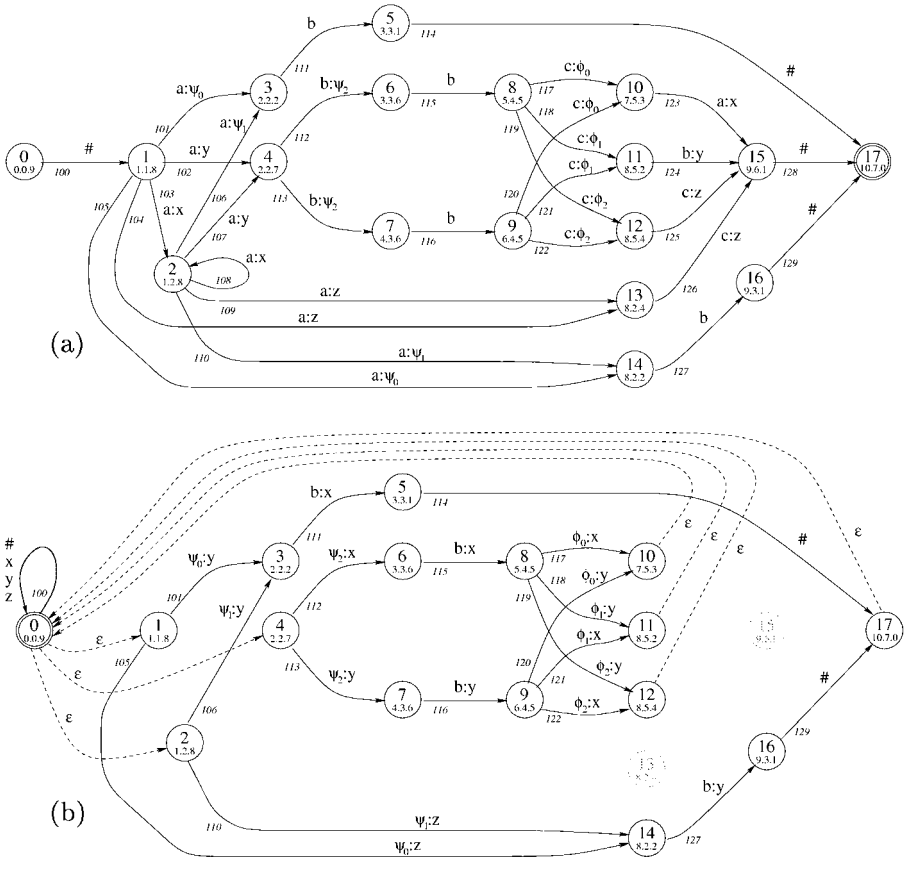


Fig. 8. Preliminary (non-minimal) factors, being (a) a functional FST T_1' and (b) an ambiguous fail-safe FST T_2' (Example 2)

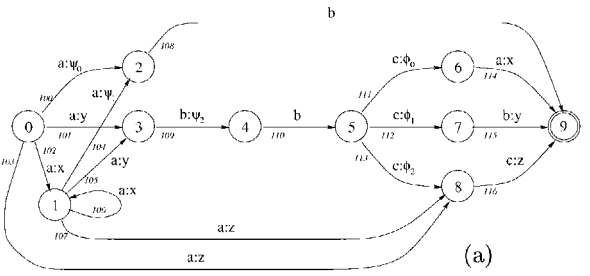
Every arc of the unfolded $T^{L,R}$ is represented in both T_1' and T_2'. Arcs that are outside any ambiguity field are copied to T_1' as they are wrt. their labels and source and destination states (Fig 8a). In T_2 they are represented by an arc looping on the initial state and labeled with the output symbol of the original arc (Fig 8b). This means, these unambiguous transductions of symbols are performed by T_1', and T_2' simply accepts the output symbols by means of looping arcs. For example, arc *102* of $T^{L,R}$ labeled with `a:y`, is copied to T_1' as it is, and a looping arc *100* labeled with `y` is created in T_2'.

All arcs of a set $\mathcal{A}$ that is inside an ambiguity field are copied to both T_1' and T_2' with their original location (regarding their source and destination) but with modified labels (Fig 8). They are copied to T_1' with their common original input symbol σ^{in} and a common intermediate symbol σ^{mid} (as output), and to T_2' with this intermediate symbol σ^{mid} (as input) and their different original output symbols σ^{out}. This causes the copy of the arc set $\mathcal{A}$ to collapse into one single arc after the minimization of T_1'. The common intermediate symbol of all arcs in $\mathcal{A}$ can be a diacritic that is unique within the whole FST, i.e., that is not used for any other arc set.

If there is concern about the size of T_1 and T_2 and their alphabets, diacritics should be used sparingly. In this case, the choice of a common σ^{mid} for a set $\mathcal{A}$ depends on the location of $\mathcal{A}$ wrt. an ambiguity field:

- At the beginning of an ambiguity field, the common intermediate symbol σ^{mid} is a *diacritic* that must be unique within the whole FST. For example, the arc set {*112, 113*} of $T^{L,R}$ gets the diacritic ψ_2, i.e., the arcs change their labels from $\mathcal{A}$={`b:x`, `b:y`} to $\mathcal{A}_1$={`b:`ψ_2, `b:`ψ_2} in T_1' and to $\mathcal{A}_2$={ψ_2`:x`, ψ_2`:y`} in T_2'. In addition, an ε-arc is inserted from the initial state of T_2' to the source state of every arc in $\mathcal{A}$, which causes the ambiguity field to begin at the initial state after minimization of T_2'.
- At a fork position that does not coincide with the beginning of an ambiguity field, the common σ^{mid} is a diacritic that needs to be unique only among all arc sets that have the same input symbol and the same input prefix set. This diacritic can be re-used with other forks. For example, the arc set {*117, 120*} gets the diacritic ϕ_0, i.e., the arcs change their labels from $\mathcal{A}$={`c:x`, `c:y`} to $\mathcal{A}_1$={`c:`ϕ_0, `c:`ϕ_0} in T_1' and to $\mathcal{A}_2$={ϕ_0`:x`, ϕ_0`:y`} in T_2'.
- In all other positions inside an ambiguity field, the common σ^{mid} equals the common input symbol σ^{in} of all arcs in a set. For example, the arc set {*115, 116*} gets the intermediate symbol `b`, i.e., the arcs change their labels from $\mathcal{A}$={`b:x`, `b:y`} to $\mathcal{A}_1$={`b`, `b`} in T_1' and keep their labels in T_2', i.e., $\mathcal{A}_2 = \mathcal{A}$.
- At the end of an ambiguity field, one of the above rules for intermediate symbols σ^{mid} is applied. In addition, an ε-arc is inserted in T_2' from the destination state of every arc in $\mathcal{A}$ to the final (= initial) state of T_2', which causes the ambiguity field to end at the final state after minimization of T_2'.

The final factors, T_1 and T_2, are obtained by replacing all boundary symbols, #, with ε, and minimizing the preliminary factors, T_1' and T_2' (Fig. 8, 9). T_1 performs a functional transduction of every accepted input string by mapping



ab	$\longrightarrow \psi_0 b$	$\longrightarrow$ { yx, zy }
$a^n aab$	$\longrightarrow x^n x \psi_1 b$	$\longrightarrow$ { $x^n xyx$, $x^n xzy$ }
$a^n abbca$	$\longrightarrow x^n y \psi_2 b \phi_0 x$	$\longrightarrow$ { $x^n yxxxx$, $x^n yyyyx$ }
$a^n abbcb$	$\longrightarrow x^n y \psi_2 b \phi_1 y$	$\longrightarrow$ { $x^n yxxyy$, $x^n yyyxy$ }
$a^n abbcc$	$\longrightarrow x^n y \psi_2 b \phi_2 z$	$\longrightarrow$ { $x^n yxxyz$, $x^n yyyxz$ }
$a^n ac$	$\longrightarrow x^n zz$	$\longrightarrow$ { $x^n zz$ }

Fig. 9. Final (minimal) factors, being (a) a functional FST T_1 and (b) an ambiguous fail-safe FST T_2 (Example 2)

every substring outside an ambiguity field to the corresponding unambiguous output, and every substring inside an ambiguity field to a unique intermediate substring. T_2 maps the former substring to itself, and the latter to a set of alternative outputs.

4 Final Remarks

An FST can contain arcs with ε (the empty string) on the input side, which is an obstacle for the above factorization. Input ε-s can be removed by removing the ε-arcs and concatenating their output symbols with the output of adjacent non-ε-arcs. This classical method, however, cannot be applied to FSTs that accept ε as input, mapping it to a non-empty string, or contain ε-loops. An ε on the output side can be handled like an ordinary symbol in factorization.

If an FST contains arcs for the *unknown symbol*, denoted by "?", in a location where a diacritic is required, factorization cannot be performed as described. For example, the arc set $\mathcal{A}=\{\texttt{?}, \texttt{?:x}\}$ cannot be factorized into $\mathcal{A}_1=\{\texttt{?:}\psi_i, \texttt{?:}\psi_i\}$ and $\mathcal{A}_2=\{\psi_i\texttt{:?}, \psi_i\texttt{:x}\}$. The first arc in $\mathcal{A}$ must map a given unknown symbol to itself which is not possible when it is factorized. The first arc in $\mathcal{A}_1$ would map an unknown symbol to ψ_i; the first arc in $\mathcal{A}_2$, however, could not map ψ_i to the same unknown symbol that occurred in the input, without using additional memory (and a special mechanism) at runtime. To solve this conflict, $\mathcal{A}$ can be factorized into two sets of arc sequences. For example, $\mathcal{A}=\{\texttt{?}, \texttt{?:x}\}$ can be factorized into $\mathcal{A}_1=\{\lceil\varepsilon\texttt{:}\psi_i, \texttt{?}\rceil, \lceil\varepsilon\texttt{:}\psi_i, \texttt{?}\rceil\}$ and $\mathcal{A}_2=\{\lceil\psi_i\texttt{:}\varepsilon, \texttt{?}\rceil, \lceil\psi_i\texttt{:}\varepsilon, \texttt{?:x}\rceil\}$. An auxiliary state is added inside every arc sequence.

The above factorization can create some redundant intermediate diacritics. A diacritic that always "co-occurs" in T_2 with another diacritic can be replaced in T_1 and T_2 by the latter without affecting the overall relation [5]. This reduces the size of the intermediate alphabet and, after minimization, the size of T_2. We mean by co-occurrence of two or more diacritics, that the arcs that are labeled

on the input side with these diacritics, have the same source, destination, and output symbol. In Example 2 (Fig. 9), ψ_1 can be replaced by ψ_0, and ϕ_2 by ϕ_1.

The algorithm described in this article has been implemented. Future research will include an experimental evaluation of the efficiency gain when processing input strings.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, USA.
2. J. Berstel. 1979. *Transductions and Context-Free Languages*. Number 38 in Leitfäden der angewandten Mathematik und Mechanik (LAMM). Studienbücher Informatik. Teubner, Stuttgart, Germany.
3. C. C. Elgot, and J. E. Mezei. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, pages 47–68, January.
4. L. Karttunen, J.-P. Chanod, G. Grefenstette, and A. Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.
5. A. Kempe. 2000. Reduction of intermediate alphabets in finite-state transducer cascades. In *Proc. 7th Conference on Automatic Natural Language Processing (TALN)*, Lausanne, Switzerland. *to appear*
6. M. Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–312.
7. C. Reutenauer, and M. P. Schützenberger. 1991. Minimization of rational word functions. *SIAM Journal of Computing*, 20(4):669–685.
8. J. Sakarovitch. 1998. A construction on finite automata that has remained hidden. *Theoretical Computer Science*, 204:205-231.
9. M. P. Schützenberger. 1961. A remark on finite transducers. *Information and Control*, 4:185–187.
10. M. P. Schützenberger. 1976. Sur les rélations rationelles entre monoïdes libres. *Theoretical Computer Science*, 3:243-259.

MONA Implementation Secrets

Nils Klarlund[1], Anders Møller[2], and Michael I. Schwartzbach[2]

[1] AT&T Labs–Research, klarlund@research.att.com
[2] BRICS, University of Aarhus, {amoeller,mis}@brics.dk

Abstract. The MONA tool provides an implementation of the decision procedures for the logics WS1S and WS2S. It has been used for numerous applications, and it is remarkably efficient in practice, even though it faces a theoretically non-elementary worst-case complexity. The implementation has matured over a period of six years. Compared to the first naive version, the present tool is faster by several orders of magnitude. This speedup is obtained from many different contributions working on all levels of the compilation and execution of formulas. We present a selection of implementation "secrets" that have been discovered and tested over the years, including formula reductions, DAGification, guided tree automata, three-valued logic, eager minimization, BDD-based automata representations, and cache-conscious data structures. We describe these techniques and quantify their respective effects by experimenting with separate versions of the MONA tool that in turn omit each of them.

1 Introduction

MONA [14,20,22] is an implementation of the decision procedures for the logics WS1S and WS2S [28]. They have long been known to be decidable [7,8,13], but with a non-elementary lower bound [21]. For many years it was assumed that this discouraging complexity precluded any useful implementations.

However, MONA has been developed at BRICS since 1994, when our initial attempt at automatic pointer analysis through automata calculations took four hours to complete. Today MONA has matured into an efficient and popular tool on which the same analysis is performed in a couple of seconds. Through those years, many different approaches have been tried out, and a good number of implementation "secrets" have been discovered. This paper describes the most important tricks we have learned, and it tries to quantify their relative merits on a number of benchmark formulas.

Of course, the resulting tool still has a non-elementary worst-case complexity. Perhaps surprisingly, this complexity also contributes to successful applications, since it is provably linked to the succinctness of the logics. If we want to describe a particular regular set, then a WS1S formula may be non-elementarily more succinct that a regular expression or a transition table.

The niche for MONA applications contains those structures that are too large and complicated to describe by other means, yet not so large as to require infeasible computations. Happily, many interesting projects fit into this niche,

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 182–194, 2001.

including hardware verification [2,3], pointer analysis [16,12], controller synthesis [25,15], natural languages [23], parsing tools [10], Presburger arithmetic [26], and verification of concurrent systems [17,1,24,27].

2 MONA, WS1S, and WS2S

The first versions of MONA were based on a logic about finite strings, the monadic second-order logic M2L(Str). In this notation, first-order variables are interpreted over the positions in a finite string. Thus, for a given interpretation, there is a maximum value that a first-order variable may take on. A second-order variable denotes a subset of positions. A formula is valid if it holds for any finite string. The decision procedure for this logic is slightly easier to implement than that of the current MONA, which is based on WS1S. This logic is simpler to explain: a first-order variable denotes a natural number, and a second-order variable denotes a finite set of numbers. Both logics allow the comparison of variables in the expected ways depending on their order: $<$, $\subseteq$, $=$, $\in$, etc. Also, a function symbol $+1$ is allowed on first-order terms. It denotes the successor (where in the case of M2L(Str), the successor of the last position is defined as the first position).

MONA additionally supports the logic WS2S with two successors. Also, there is explicit syntax for Presburger constants. Finally, it implements the variation WSRT which allows values of recursive data types rather than simply binary trees. The MONA manual [20] describes the syntax and semantics of the MONA language and the features of the tool.

The automaton for a formula is constructed recursively from automata representing subformulas. In the cases of M2L(Str) and WS1S, each automaton describes a language of strings over the alphabet $\{0,1\}^k$, where k is the number of free variables in the subformula. Each string represents an interpretation, that is, an assignment of values to variables that are free in the subformula; the language is the set of strings that define satisfying interpretations. For WS2S this is generalized to tree automata.

3 Benchmark Formulas

The experiments presented in the following section are based on twelve benchmark formulas, here shown with their sizes, the logics they are using, and their time and space consumptions when processed by MONA 1.4 (on a 296MHz UltraSPARC with 1GB RAM):

Benchmark	Name	Size	Logic	Time	Space
A	`dflipflop.mona`	2 KB	WS1S (M2L-Str)	0.4 sec	3 MB
B	`euclid.mona`	6 KB	WS1S (Presburger)	33.1 sec	217 MB
C	`fischer_mutex.mona`	43 KB	WS1S	15.1 sec	13 MB
D	`html3_grammar.mona`	39 KB	WS2S (WSRT)	137.1 sec	208 MB
E	`lift_controller.mona`	36 KB	WS1S	8.0 sec	15 MB
F	`mcnc91_bbsse.mona`	9 KB	WS1S	13.2 sec	17 MB
G	`reverse_linear.mona`	11 KB	WS1S (M2L-Str)	3.2 sec	4 MB
H	`search_tree.mona`	19 KB	WS2S (WSRT)	30.4 sec	5 MB
I	`sliding_window.mona`	64 KB	WS1S	40.3 sec	59 MB
J	`szymanski_acc.mona`	144 KB	WS1S	20.6 sec	9 MB
K	`von_neumann_adder.mona`	5 KB	WS1S	139.9 sec	116 MB
L	`xbar_theory.mona`	14 KB	WS2S	136.4 sec	518 MB

The benchmarks have been picked from a large variety of MONA applications ranging from hardware verification to encoding of natural languages.

`dflipflop.mona` – a verification of a D-type flip-flop circuit [3]. Provided by Abdel Ayari.

`euclid.mona` – an encoding in Presburger arithmetic of six steps of reachability on a machine that implements Euclid's GCD algorithm [26]. Provided by Tom Shiple.

`fischer_mutex.mona` and `lift_controller.mona` – duration calculus encodings of Fischer's mutual exclusion algorithm and a mine pump controller, translated to MONA code [24]. Provided by Paritosh Pandya.

`html3_grammar.mona` – a tree-logic encoding of the HTML 3.0 grammar annotated with 10 parse-tree formulas [10]. Provided by Niels Damgaard.

`reverse_linear.mona` – verifies correctness of a C program reversing a pointer-linked list [16].

`search_tree.mona` – verifies correctness of a C program deleting a node from a search tree [12].

`sliding_window.mona` – verifies correctness of a sliding window network protocol [27]. Provided by Mark Smith.

`szymanski_acc.mona` – validation of the parameterized Szymanski problem using an accelerated iterative analysis [5]. Provided by Mamoun Filali-Amine.

`von_neumann_adder.mona` and `mcnc91_bbsse.mona` – verification of sequential hardware circuits; the first verifies that an 8-bit von Neumann adder is equivalent to a standard carry-chain adder, the second is a benchmark from MCNC91 [29]. Provided by Sebastian Mödersheim.

`xbar_theory.mona` – encodes a part of a theory of natural languages in the Chomsky tradition. It was used to verify the theory and led to the discovery of mistakes in the original formalization [23]. Provided by Frank Morawietz.

We will use these benchmarks to illustrate the effects of the various implementation "secrets" by comparing the efficiency of MONA shown in the table above with that obtained by handicapping the MONA implementation by not using the techniques.

4 Implementation Secrets

The MONA implementation has been developed and tuned over a period of six years. Many large and small ideas have contributed to a combined speedup of several orders of magnitude. Improvements have taken place at all levels, which we illustrate with the following seven examples from different phases of the compilation and execution of formulas.

To enable comparisons, we summarize the effect of each implementation "secret" by a single dimensionless number for each benchmark formula. Usually, this is simply the speedup factor, but in some cases where the numerator is not available, we argue for a more synthetic measure. If a benchmark cannot run on our machine, it is assigned time ∞.

4.1 Eager Minimization

When MONA inductively translates formulas to automata, a Myhill-Nerode minimization is performed after every product and projection operation. Naturally, it is preferable to operate with as small automata as possible, but our strategy may seem excessive since minimization often exceeds 50% of the total running time. This suspicion is strengthened by the fact that MONA automata by construction contain only reachable states; thus, minimization only collapses redundant states.

Three alternative strategies to the eager one currently used by MONA would be to perform only the very final minimization, only the ones occurring after projection operations, or only the ones occurring after product operations. Many other heuristics could of course also be considered. The following table results from such an investigation:

Benchmark	Time				Effect
	Only final	After project	After product	Always	
A	∞	∞	0.6 sec	0.4 sec	1.5
B	∞	∞	∞	33.1 sec	∞
C	∞	∞	32.3 sec	15.1 sec	2.1
D	∞	∞	290.6 sec	137.1 sec	2.1
E	∞	∞	19.4 sec	8.0 sec	2.4
F	∞	∞	36.7 sec	13.2 sec	2.8
G	∞	∞	5.8 sec	3.2 sec	1.8
H	∞	∞	59.6 sec	30.4 sec	2.0
I	∞	∞	74.4 sec	40.3 sec	1.8
J	∞	∞	36.3 sec	20.6 sec	1.8
K	∞	∞	142.3 sec	139.9 sec	1.0
L	∞	∞	∞	136.4 sec	∞

"Only final" is the running time when minimization is only performed as the final step of the translation; "After project" is the running time when minimization is also performed after every projection operation; "After product" is the running

time when minimization is instead performed after every product operation; "Always" is the time when minimization is performed eagerly; and "Effect" is the "After product" time compared to the "Always" time (since the other two strategies are clearly hopeless). Eager minimization is seen to be always beneficial and in some cases essential for the benchmark formulas.

4.2 Guided Tree Automata

Tree automata are inherently more computationally expensive because of their three-dimensional transition tables. We have used a technique of factorization of state spaces to split big tree automata into smaller ones. The basic idea, which may result in exponential savings, is explained in [4]. To exploit this feature, the MONA programmer must manually specify a *guide*, which is a top-down tree automaton that assigns state spaces to the nodes of a tree. However, when using the WSRT logic, a canonical guide is automatically generated. For our two WSRT benchmarks, we measure the effect of this canonical guide:

Benchmark	Without guide	With guide	Effect
D	584.0 sec	137.1 sec	4.3
H	∞	30.4 sec	∞

"Without guide" shows the running time without any guide, while "With guide" shows the running time with the canonical WSRT guide; "Effect" shows the "Without guide" time compared to the "With guide" time. We have only a small sample space here, but clearly guides are very useful. This is hardly surprising, since they may yield an asymptotic improvement in running time.

4.3 Cache-Conscious Data Structures

The data structure used to represent the BDDs for transition functions has been carefully tuned to minimize the number of cache misses that occur. This effort is motivated in earlier work [18], where it is determined that the number of cache misses during unary and binary BDD apply steps totally dominates the running time.

In fact, we argued elsewhere [18] that if A1 is the number of unary apply steps and A2 is the number of binary apply steps, then there exists constant m, c_1, and c_2 such that the total running time is approximately $m(c_1 \cdot \mathrm{A1} + c_2 \cdot \mathrm{A2})$. Here, m is the machine dependent delay incurred by an L2 cache miss, and c_1 and c_2 are the average number of cache misses for unary and binary apply steps. This estimate is based the assumption that time incurred for manipulating auxiliary data structures, such as those used for describing subsets in the determinization construction, is insignificant. For the machine we have used for experiments, it is by a small C utility determined that $m = 0.43\mu s$. In our BDD implementation, explained in [18], we have estimated from algorithmic considerations that $c_1 = 1.7$ and $c_2 = 3$ (the binary apply may entail the use of unary apply steps for doubling tables that were too small—these steps are not counted towards the

time for binary apply steps, and that is why we can use the figure $c_2 = 3$); we also estimated that for an earlier conventional implementation, the numbers were $c_1 = 6.7$ and $c_2 = 7.3$. The main reason for this difference is that our specialized package stores nodes directly under their hash address to minimize cache misses; traditional BDD packages store BDD nodes individually with the hash table containing pointers to them—roughly doubling the time it takes to process a node. We no longer support the conventional BDD implementation, so to measure the effect of cache-consciousness, we must use the above formula to estimate the running times that would have been obtained today.

In the following experiment, we have instrumented MONA to obtain the exact numbers of apply steps:

Benchmark	Apply1	Apply2	Misses	Auto	Predicted	Conventional	Effect
A	183,949	28,253	397,472	0.2 sec	0.2 sec	0.6 sec	3.0
B	21,908,722	3,700,856	48,347,395	32.8 sec	20.8 sec	74.7 sec	3.6
C	24,585,292	1,428,381	46,080,139	14.2 sec	19.8 sec	75.2 sec	3.8
E	9,847007	822,796	19,208,299	7.7 sec	8.2 sec	30.9 sec	3.8
F	13,406,047	5,717,453	39,942,638	12.8 sec	17.2 sec	56.6 sec	3.3
G	233,566	54,814	561,504	0.5 sec	0.3 sec	0.8 sec	2.7
I	36,629,195	11,153,733	95,730,831	37.0 sec	41.2 sec	140.5 sec	3.4
J	10,497,759	2,257,791	24,619,563	11.6 sec	10.6 sec	37.3 sec	3.5
K	129,126,447	10,485,623	250,971,828	137.4 sec	107.9 sec	404.7 sec	3.8

"Apply1" is the number of unary apply steps; "Apply2" is the number of binary apply steps; "Misses" is the number of cache misses predicted by the formula above; "Auto" is the part of the actual running time involved in automata constructions; "Predicted" is the running time predicted from the cache misses alone; "Conventional" is the predicted running time for a conventional BDD implementation that was not cache-conscious; and "Effect" is "Conventional" compared to "Predicted". In most cases, the actual running time is close to the predicted one (within 25%). Note that there are instances where the actual time is about 50% larger than the estimated time: benchmark B involves a lengthy subset construction on an automaton with small BDDs—thus it violates the assumption that the time handling accessory data structures is insignificant; similarly, benchmark G also consists of many automata with few BDD nodes prone to violating the assumption.

In an independent comparison [26] it was noted that MONA was consistently twice as fast as a specially designed automaton package based on a BDD package considered efficient. In [18], the comparison to a traditional BDD package yielded a factor 5 speedup.

4.4 BDD-Based Automata Representation

Its reasonable to ask: "What would happen if we had simply represented the transition tables in a standard fashion, that is, a row for each state and a column for each letter?". Under this point of view, it makes sense to define a letter for each bit-pattern assignment to the free variables of a subformula (as opposed

to the larger set of all variables bound by an outer quantifier). We have instrumented MONA to measure the sum of the number of entries of all such automata transition tables constructed during a run of a version of MONA without BDDs:

Benchmark	Misses	Table entries	Effect
A	397,472	237,006	0.6
B	48,347,395	2,973,118	0.1
C	46,080,139	1,376,499,745,600	29,871.9
E	19,208,299	290,999,305,488	15,149.7
F	39,942,638	2,844,513,432,416,357,974,016	71,214,961,626,128.9
G	561,202	912,194	1.6
I	95,730,831	116,387,431,997,281,136	1,215,777,934.7
J	24,619,563	15,424,761,908	626.5
K	250,971,828	2,544,758,557,238,438	10,139,618.4

"Misses" is again the number of cache misses in our BDD-based implementation, and "Table entries" is the total number of table entries in the naive implementation. To roughly estimate the effect of the BDD-representation, we conservatively assume that each table entry results in just a single cache miss; thus, "Effect" compares "Table entries" to "Misses". The few instances where the effect is less than one correctly identify benchmark formulas where the BDDs are less necessary, but are also artifacts of our conservative assumption. Conversely, the extremely high effects are associated with formulas that could not possibly be decided without BDDs. Of course, the use of BDD-structures completely dominates all other optimizations, since no implementation could realistically be based on the naive table representation.

The BDD-representation was the first breakthrough of the MONA implementation, and the other "secrets" should really be viewed with this as baseline. The first implementation did not actually use tables but a conjunctive normal form. Nevertheless, the effect of switching to BDDs was stunning.

4.5 DAGification

Internally, MONA is divided into a front-end and a back-end. The front-end parses the input and builds a data structure representing the automata-theoretic operations that will calculate the resulting automaton. The back-end then inductively carries out these operations.

The generated data structure is often seen to contain many common subformulas. This is particularly true when they are compared relative to *signature equivalence*, which holds for two formulas ϕ and ϕ' if there is an order-preserving renaming of the variables in ϕ (increasing with respect to the indices of the variables) such that the representations of ϕ and ϕ' become identical.

A property of the BDD representation is that the automata corresponding to signature-equivalent trees are isomorphic in the sense that only the node indices differ. This means that intermediate results can be reused by simple exchanges of node indices. For this reason, MONA represents the formulas in a DAG (Directed

Acyclic Graph), not a tree. The DAG is conceptually constructed from the tree using a bottom-up collapsing process, based on the signature equivalence relation as described in [11].

Clearly, constructing the DAG instead of the tree incurs some overhead, but the following experiments show that the benefits are significantly larger:

Benchmark	Nodes		Time		Effect
	Tree	DAG	Tree	DAG	
A	2,532	296	1.7 sec	0.4 sec	4.3
B	873	259	79.2 sec	33.1 sec	2.4
C	5,432	461	40.1 sec	15.1 sec	2.7
D	3,038	270	∞	137.1 sec	∞
E	4,560	505	20.5 sec	8.0 sec	2.6
F	1,997	505	49.1 sec	13.2 sec	3.7
G	56,932	1,199	∞	3.2 sec	∞
H	8,180	743	∞	30.4 sec	∞
I	14,058	1,396	107.1 sec	40.3 sec	2.7
J	278,116	6,314	∞	20.6 sec	∞
K	777	273	284.0 sec	139.9 sec	2.0
L	1,504	388	∞	136.4 sec	∞

"Nodes" shows the number of nodes in the representation of the formula. "Tree" is the number of nodes using an explicit tree representation, while "DAG" is the number of nodes after DAGification. "Time" shows the running times for the same two cases. "Effect" shows the "Tree" running time compared to the "DAG" running time. The DAGification is seen to provide a substantial and often essential gain in efficiency.

The effects reported sometimes benefit from the fact that the restriction technique presented in the following subsection knowingly generates redundant formulas. This explains some of the failures observed.

4.6 Three-Valued Logic and Automata

In earlier versions of MONA, we struggled with the issue of encoding first-order variables as second-order variables—that's the standard technique for monadic second-order logics, but it raises the issue of *restrictions*: the common phenomenon that a formula ϕ makes sense, relative to some exterior conditions, only when an associated restriction holds. The restriction is also a formula, and the main issue is that ϕ is now essentially undefined outside the restriction. Later, when we chose to base MONA on WS1S instead of a monadic second-order logic on strings, we sometimes encountered state space explosions when we constrained variables in order to emulate the string-based semantics.

The nature of these problems is very technical, but fortunately they can be solved through a theory of restriction couched in a three-valued logic [19]. Under this view, a *restricted subformula* ϕ is associated with a restriction ϕ_R different from *true*; an *unrestricted formula* is associated with a restriction ϕ_R that is

true. We do not outline the theory of restrictions here, except for noting that restriction of a conjunction (or disjunction) is the conjunction of the restrictions of the conjuncts (or disjuncts).

According to [19], we can guarantee that the WS1S framework handles all formulas written in the earlier string logic, even with intermediate automata that are no bigger than when run through the original decision procedure. Also, the running time of the original procedure may be asymptotically worse than with the WS1S formulation. Unfortunately, there is no way of disabling this feature to provide a quantitative comparison.

4.7 Formula Reductions

Formula reduction is a means of "optimizing" the formulas in the DAG before translating them into automata. The reductions are based on a syntactic analysis that attempts to identify valid subformulas and equivalences among subformulas.

There are some non-obvious choices here. How should computation resources be apportioned to the reduction phase and to the automata calculation phase? Must reductions guarantee that automata calculations become faster? Should the two phases interact? Our answers are based on some trial and error along with some provisions to cope with subtle interactions with other of our optimization secrets.

MONA 1.4 performs three kinds of formula reductions: 1) simple equality and boolean reductions, 2) special quantifier reductions, and 3) special conjunction reductions. The first kind can be described by simple rewrite rules (only some typical ones are shown):

$$x = x \rightsquigarrow \mathit{true} \qquad \phi \wedge \phi \rightsquigarrow \phi$$
$$\mathit{true} \wedge \phi \rightsquigarrow \phi \qquad \neg\neg\phi \rightsquigarrow \phi$$
$$\mathit{false} \wedge \phi \rightsquigarrow \mathit{false} \qquad \neg\mathit{false} \rightsquigarrow \mathit{true}$$

These rewrite steps are guaranteed to reduce complexity, but will not cause significant improvements in running time, since they all either deal with constant size automata or rarely apply in realistic situations. Nevertheless, they are extremely cheap, and they may yield small improvements, in particular on machine generated MONA code.

The second kind of reductions can potentially cause tremendous improvements. The non-elementary complexity of the decision procedure is caused by the automaton projection operations, which stem from quantifiers. The accompanying determinization construction may cause an exponential blow-up in automaton size. Our basic idea is to apply a rewrite step resembling *let*-reduction, which removes quantifiers:

$$\exists X : \phi \rightsquigarrow \phi[T/X] \quad \text{provided that } \phi \Rightarrow X = T \text{ is valid, and } T \text{ is some term satisfying } FV(T) \subseteq FV(\phi)$$

where $FV(\cdot)$ denotes the set of free variables. For several reasons, this is not the way to proceed in practice. First of all, finding terms T satisfying the side

condition can be an expensive task, in worst case non-elementary. Secondly, the translation into automata requires the formulas to be "flattened" by introduction of quantifiers such that there are no nested terms. So, if the substitution $\phi[T/X]$ generates nested terms, then the removed quantifier is recreated by the translation. Thirdly, when the rewrite rule applies in practice, ϕ usually has a particular structure as reflected in the following more restrictive rewrite rule chosen in MONA:

$$\exists X : \phi \rightsquigarrow \phi[Y/X] \quad \text{provided that } \phi \equiv \cdots \wedge X = Y \wedge \cdots, \text{ and } Y \text{ is some variable other than } X$$

In contrast to equality and boolean reductions, this rule is not guaranteed to improve performance, since substitutions may cause the DAG reuse degree to decrease.

The third kind of reductions applies to conjunctions, of which there are two special sources. One is the formula flattening just mentioned; the other is the formula restriction technique mentioned in Section 4.6. Both typically introduce many new conjunctions. Studies of a graphical representation of the formula DAGs (MONA can create such graphs automatically) led us to believe that many of these new conjunctions are redundant. A typical rewrite rule addressing such redundant conjunctions is the following:

$$\phi_1 \wedge \phi_2 \rightsquigarrow \phi_1 \quad \text{provided that } \mathit{unrestr}(\phi_2) \subseteq \mathit{unrestr}(\phi_1) \cup \mathit{restr}(\phi_1) \text{ and } \mathit{restr}(\phi_2) \subseteq \mathit{restr}(\phi_1)$$

Here, $\mathit{unrestr}(\phi)$ is the set of unrestricted conjuncts in ϕ, and $\mathit{restr}(\phi)$ is the set of restricted conjuncts in ϕ. This reduction states that it is sufficient to assert ϕ_1 when $\phi_1 \wedge \phi_2$ was originally asserted in situations where the unrestricted conjuncts of ϕ_2 are already conjuncts of ϕ_1—whether restricted or not—and the restricted conjuncts of ϕ_2 are unrestricted conjuncts of ϕ_1. It is not sufficient that they be restricted conjuncts of ϕ_1, since the restrictions may not be the same in ϕ_1.

With the DAG representation of formulas, the reductions just described can be implemented relatively easily in MONA. The table below shows the effects of performing the reductions on the benchmark formulas:

Benchmark	Hits			Time					Effect
	Simple	Quant.	Conj.	None	Simple	Quant.	Conj.	All	
A	12	8	22	0.8 sec	0.7 sec	0.7 sec	0.7 sec	0.4 sec	2.0
B	10	45	0	58.2 sec	58.8 sec	56.2 sec	56.8 sec	33.1 sec	1.8
C	9	13	8	43.7 sec	41.9 sec	37.1 sec	42.9 sec	15.1 sec	2.9
D	4	28	27	542.7 sec	536.1 sec	296.0 sec	404.7 sec	137.1 sec	4.0
E	5	6	19	22.6 sec	23.4 sec	16.6 sec	22.7 sec	8.0 sec	2.8
F	3	1	1	28.3 sec	29.9 sec	27.0 sec	27.2 sec	13.2 sec	2.1
G	65	318	191	6.1 sec	5.9 sec	6.1 sec	5.9 sec	3.2 sec	1.9
H	35	32	81	104.1 sec	102.6 sec	71.0 sec	98.5 sec	30.4 sec	3.4
I	102	218	7	76.2 sec	76.5 sec	75.0 sec	76.0 sec	40.3 sec	1.9
J	91	0	1	37.3 sec	37.9 sec	37.6 sec	37.0 sec	20.6 sec	1.9
K	9	4	1	313.7 sec	267.9 sec	240.3 sec	302.6 sec	139.9 sec	2.3
L	4	4	18	∞	∞	∞	∞	136.4 sec	∞

"Hits" shows the number of times each of the three kinds of reduction is performed; "Time" shows the total running time in the cases where no reductions are performed, only the first kind of reductions are performed, only the second, only the third, and all of them together. "Effect" shows the "None" times compared to the "All" times. All benchmarks gain from formula reductions, and in a single example this technique is even necessary. Note that most often all three kinds of reductions must act in unison to obtain significant effects.

A general benefit from formula reductions is that tools generating MONA formulas from other formalisms may generate naive and voluminous output while leaving optimizations to MONA. In particular, tools may use existential quantifiers to bind terms to fresh variables, knowing that MONA will take care of the required optimization.

5 Future Developments

Several of the techniques described in the previous section can be further refined of course. The most promising ideas seem however to concentrate on the BDD representation. In the following, we describe three such ideas.

It is a well-known fact [6] that the ordering of variables in the BDD automata representation has a strong influence on the number of BDD nodes required. The impact of choosing a good ordering can be an exponential improvement in running times. Finding the optimal ordering is an NP-complete problem, but we plan to experiment with the heuristics that have been suggested [9].

We have sometimes been asked: "Why don't you encode the states of the automata in BDDs, since that is a central technique in model checking?". The reason is very clear: there is no obvious structure to the state space in most cases that would lend itself towards an efficient BDD representation. For example, consider the consequences of a subset construction or a minimization construction, where similar states are collapsed; in either case, it is not obvious how to represent the new state. However, the ideas are worth investigating.

For our tree automata, we have experimentally observed that the use of guides produce a large number of component automata many of which are almost identical. We will study how to compress this representation using a BDD-like global structure.

6 Conclusion

The presented techniques reflect a lengthy Darwinian development process of the MONA tool in which only robust and useful ideas have survived. We have not mentioned here the many ideas that failed or were surpassed by other techniques. Our experiences confirm the maxim that optimizations must be carried out at all levels and that no single silver bullet is sufficient. We are confident that further improvements are still possible.

Many people have contributed to earlier versions of MONA, in particular we are grateful to David Basin, Morten Biehl, Jacob Elgaard, Jesper Gulmann, Jacob Jensen, Michael Jørgensen, Bob Paige, Theis Rauhe, and Anders Sandholm. We also thank the MONA users who kindly provided the benchmark formulas.

References

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Computer Aided Verification, CAV '99*, volume 1633 of *LNCS*, 1999.

[2] David Basin and Nils Klarlund. Hardware verification using monadic second-order logic. In *Computer Aided Verification, CAV '95*, volume 939 of *LNCS*, 1995.

[3] David Basin and Nils Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods in Systems Design*, 13(3):255–288, 1998.

[4] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *WIA '96*, volume 1260 of *LNCS*, 1996.

[5] Jean-Paul Bodeveix and Mamoun Filali. FMona: a tool for expressing validation techniques over infinite state systems. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, volume 1785 of *LNCS*, 2000.

[6] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

[7] J.R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[8] J.R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. Internat. Cong. on Logic, Methodol., and Philos. of Sci.* Stanford University Press, 1962.

[9] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 417–420, 1991.

[10] Niels Damgaard, Nils Klarlund, and Michael I. Schwartzbach. YakYak: Parsing with logical side constraints. In *Proceedings of DLT'99*, 1999.

[11] Jacob Elgaard, Nils Klarlund, and Anders Møller. Mona 1.x: New techniques for WS1S and WS2S. In *Computer Aided Verification, CAV '98*, volume 1427 of *LNCS*, 1998.

[12] Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proceedings of European Symposium on Programming Languages and Systems*, volume 1782 of *LNCS*, 2000.

[13] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

[14] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95*, volume 1019 of *LNCS*, 1996.

[15] Thomas Hune and Anders Sandholm. A case study on using automata in control synthesis. In *Proceedings of FASE'00*, 2000.

[16] J.L. Jensen, M.E. Jørgensen, N. Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN '97 Conference on Programming Language Design and Implementation,*, pages 226–234, 1997.

[17] N. Klarlund, M. Nielsen, and K. Sunesen. A case study in automated verification based on trace abstractions. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification, The RPC-Memory Specification Case Study*, volume 1169 of *LNCS*, pages 341–374. Springer Verlag, 1996.

[18] N. Klarlund and T. Rauhe. BDD algorithms and cache misses. Technical report, BRICS Report Series RS-96-5, Department of Computer Science, University of Aarhus, 1996.

[19] Nils Klarlund. A theory of restrictions for logics and automata. In *Computer Aided Verification, CAV '99*, volume 1633 of *LNCS*, 1999.

[20] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998. URL: `http://www.brics.dk/mona/manual.html`.

[21] A.R. Meyer. Weak monadic second-order theory of successor is not elementary recursive. In R. Parikh, editor, *Logic Colloquium, (Proc. Symposium on Logic, Boston, 1972)*, volume 453 of *LNCS*, pages 132–154, 1975.

[22] Anders Møller. MONA homepage. URL: `http://www.brics.dk/mona/`.

[23] Frank Morawietz and Tom Cornell. The logic-automaton connection in linguistics. In *Proceedings of LACL 1997*, volume 1582 of *LNAI*, 1997.

[24] Paritosh K. Pandya. DCVALID 1.2. Technical report, Theoretical Computer Science Group, Tata Institute of Fundamental Research, 1997.

[25] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for web services. In *FASE*, volume 1382 of *LNCS*, 1998.

[26] Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *CAV*, volume 1427 of *LNCS*, 1998.

[27] Mark A. Smith and Nils Klarlund. Verification of a sliding window protocol using IOA and Mona, 1999. Submitted for publication.

[28] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[29] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0. In *Tech. Rep. MCNC*, 1991.

Cursors

Vincent Le Maout

Institut Gaspard Monge, Université de Marne La Vallée*
Champs-sur-Marne, France
lemaout@univ-mlv.fr

Abstract. The iterator concept is becoming the fundamental abstraction of reusable software and the key to modularity and clean code especially in object-oriented languages like C++ and Java. They serve as accessors to a sequence hiding the implementation details from the algorithm and their encapsulation power allows true generic programming. The Standard Template Library defines clearly their behavior on simple sequences like linked lists or vectors. In this paper, we define the concept of cursor which can be seen as a generalization of the iterator concept to more complex data structures than sequences, in this case acyclic automata. We show how elegant and efficient they can be on applications written in C++ and based on the Automaton Standard Template Library.

1 Introduction

Cursors introduce a software layer between the deterministic finite automaton classes of the Automaton Standard Template Library (ASTL, an automaton library written in C++ [VLM98]) and the algorithms. Likewise the iterator concept, the cursor concept serves two purposes: making the algorithm undependant from the automaton structure and removing some of the algorithmic responsibilities from the algorithm core. It must be regarded as a generalization of the iterator concept: the main difference is that when iterating on data, a cursor needs to know which path to follow whereas an iterator knows of only one. Therefore, assigning a traversal algorithm to a cursor makes it an iterator and a cursor can be built from an iterator too.
This obviously implies that these objects have a well-defined, consistent and simple behavior to rely on. Moreover, generic programming standards impose tough efficiency constraints one has to comply to.
This paper introduces the concept of cursor on acyclic automata. We first present a few definitions and programming abstractions and then expose the main obstacles we met using ASTL that required the introduction of three major models of cursor: forward, stack and depth-first cursors. We then discuss the issue of applying algorithms and show how easy and straightforward it is to combine them to create more powerful algorithms.

* Supported by Lexiquest SA, http://www.lexiquest.com/

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 195–207, 2001.

2 Definitions

2.1 Deterministic Finite Automaton

To allow more flexibility, we add to the classical DFA definition a set of *tags*, that is any data needed to apply an algorithm. The set τ maps each state to a tag.
Let $A(\Sigma, Q, i, F, \Delta, T, \tau)$ be a 7-uple defined as follow: Σ is the alphabet, Q a set of states, $i \in Q$ the initial state, $F \subseteq Q$ a set of final (accepting) states, $\Delta \subseteq Q \times \Sigma \times Q$ a set of transitions, T a set of tags, $\tau \subset Q \times T$ a relation from states to tags.
We distinguish one special state noted 0 and called the null or sink state. For every automaton $A(\Sigma, Q, i, F, \Delta, T, \tau)$ we have $0 \in Q$. The language of a DFA is written $L(A)$.
We define $P(X)$ as the power-set of a set X.

2.2 Access Functions and Sink Transitions

To access Δ we define two transition functions δ_1 and δ_2 :

$$\delta_1 \ : \ Q \times \Sigma \to Q$$
$$\forall q \in Q, \forall a \in \Sigma, \ \delta_1(q,a) = \begin{cases} p \text{ if } (q,a,p) \in \Delta \\ 0 \text{ otherwise} \end{cases}$$

δ_1 retrieves a transition target given the source state and a letter. In the case of undefined transitions the result is the null (sink) state.
A transition verifying the following property is called a *sink transition*:

$$(q,a) \in Q \times \Sigma, \ q = 0 \text{ or } \delta_1(q,a) = 0$$

δ_2 retrieves the set of all outgoing transitions of a source state allowing thus its traversal :

$$\delta_2 \ : \ Q \to P(\Sigma \times Q)$$
$$\forall q \in Q, \ \delta_2(q) = \{(a,p) \in \Sigma \times Q \text{ such that } (q,a,p) \in \Delta\}$$

2.3 Concepts and Models

We call a *concept* a set of requirements on a type covering three aspects:

1. The interface (the methods signatures).
2. The methods semantics (the behavior).
3. The methods complexities (the computing time).

We call *model* of a concept a type of object conforming to this concept. Concepts are usually abstract classes and models are concrete types implementing the concept. For instance, a C-like pointer is a model of iterator: it provides an operator `[ ]` returning a reference on the i^{nth} element of a sequence in constant time.

2.4 Object Properties

1. An object has a *singular value* when it only guarantees assignment operation and results of most expressions are undefined. For example, an uninitialized pointer has a singular value.
2. We say that an object `x` is *assignable* iff it defines an operator `=` allowing assignment from an object `y` of the same type. The postcondition of the assignment "`x = y`" is "`x` is a copy of `y`".
3. An object is *default constructible* iff no values are needed to initialize it. In C++, it defines a default constructor.
4. An object is *equality-comparable* iff it provides a way to compare itself with other objects of the same type. In C++, such an object defines an operator `==` returning a boolean value.
5. An object is *less-than-comparable* iff there exist a partial order relation on the objects of its type. In C++, such an object provides an operator `<` returning a boolean value.

2.5 Iterators

Quoting from SGI Standard Template Library reference documentation [SGI99]:

> Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.

Iterators are:

1. default constructible.
2. assignable (infix operator `=`).
3. singular if none of the following properties is true. An iterator with a singular value only guarantees assignment operation.
4. incrementable if applying `++` operator leads to a well-defined position.
5. dereferenceable if pointed object can be safely retrieved (prefix operator `*`).
6. equality-comparable (infix operator `==`).

Iterators constitute the link between the algorithm and the underlying data structure: they provide the sufficient level of encapsulation to make processing undependant from the data.

2.6 Range

A valid *range* `[x,y)` where `x` and `y` are iterators represents a set of positions from `x` to `y` with the following properties :

1. `[x,y)` refers to all positions between `x` and `y` but not including `y` which is called the *end-of-range* iterator (also denoted as a *past-the-end* iterator).
2. All iterators in the range except `y` are incrementable and dereferenceable.
3. All iterators including `y` are equality comparable.
4. A finite sequence of incrementations of `x` leads to position `y`.

3 Weaknesses of the Two-Layer Model

ASTL was structured around a "two-layer" model with algorithms directly lying on the automaton data structure (Fig. 3).

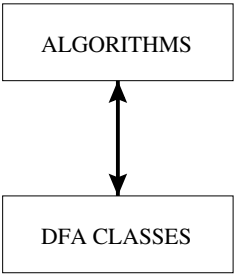


Fig. 1. Former ASTL structure

This unfortunately induces limits to the algorithm application spectrum and shortcomings regarding mainly four aspects:

1. A much too strong coupling between the algorithms and the automaton classes reduces genericity by enforcing strictly complying input data: a new algorithm version has to be written for more exotic data or a new entire automaton class has to be designed (which is not always possible due to combinatorial problems for instance). This yields multiple instances of the same code which contradicts generic programming purpose. Moreover, an intermediary layer allows for data hiding and algorithm application to other structures than automata.
2. Common parts shared by many algorithms should be written only once and be reused as is. This means moving intensively used functionalities to a third party other than algorithm core or automaton class. The best example is the iteration over transitions of a DFA: all algorithms have an traversal policy and they can be reduced to a few ones, depth-first, breadth-first and a couple of others. A cursor is the place to implement these common parts.
3. The algorithm must not impose a behavior that should be externally decided of: for example, writing a DFA union algorithm should not involve decisions about applying it on-the-fly, by lazy construction or by copy. The decision should be taken at utilization time and not at design time (see how to apply an algorithm in Sect. 7).
4. Reusing and combining algorithms should be a straightforward operation requiring no extra code. Hard coded algorithms prevent such flexibility but cursor adapters allow it.

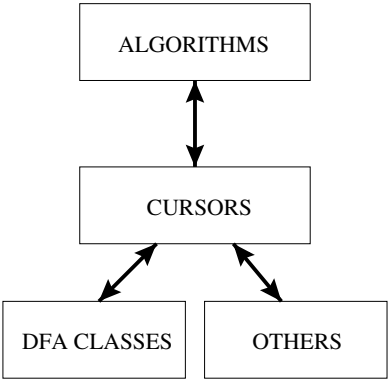


Fig. 2. Current ASTL structure

4 Forward Cursor

Using a cursor is a way of entirely hiding the processed data structure which needs not to actually be an automaton any more (Fig. 2) . Moreover, it allows to move functionalities from the algorithm to the cursor layer leading to clearer and lighter algorithms (see algorithm language in sect. 6.5).

4.1 Definition

A forward cursor is basically an object pointing to a DFA transition $(q, a, p) \in Q \times \Sigma \times Q$ (possibly a sink transition) allowing access and forward moves along it. Likewise iterators they represent positions in the automaton and therefore can be used to define ranges over it.

4.2 Properties

1. A forward cursor is default constructible but has by default a singular value.
2. A forward cursor is assignable.
3. A forward cursor is equality-comparable.
4. A forward cursor is dereferenceable (one can access to source, target and letter of pointed transition) iff it does not point to a sink transition.
5. A forward cursor is "incrementable" (it may move along the currently pointed transition) iff it does not point to a sink transition.

4.3 Interface

In the following table, we will write:

`X`	for a type which is a model of forward cursor
`x, y`	for objects of type `X`
`a`	for a letter (an unsigned integral type)
$(q, a, p) \in Q \times \Sigma \times Q$	for the transition `x` is pointing to

Name	Expression	Semantics
source state	`x.src();`	return q
aim state	`x.aim();`	return p
letter	`x.letter();`	return a
final source	`x.src_final();`	return `true` if $q \in F$
final aim	`x.aim_final();`	return `true` if $p \in F$
comparison	`(x == y)`	`true` if `x` points on the same transition as `y`
forward	`x.forward();`	move along currently pointed transition
forward with letter	`x.forward(a);`	move along transition labeled with `a` return `true` if $\delta_1(q, a) \neq 0$
first transition	`x.first_transition();`	set `x` on the first transition of $\delta_2(q)$ return `true` if $\delta_2(q) \neq \emptyset$
next transition	`x.next_transition();`	move on to the next transition of set $\delta_2(q)$. return `false` if reached transition (q, a', p') is a sink transition
find	`x.find(a);`	set `x` on the transition labeled with letter `a`. return `true` if $(q, a, \delta_1(q, a))$ is not a sink transition

4.4 Using Cursors and Their Adaptability Power

Here is an example of algorithm testing if a word defined by a range `[first,last)` is in the recognized language of a DFA accessed through a cursor `c`:

```
bool is_in(Iterator first, Iterator last, ForwardCursor c) {
  while (first != last && c.forward(*first))
    ++first;
  return first == last && c.src_final();
}
```

Any complying cursor can be passed to this algorithm: the default forward cursor or any cursor adapter implementing a different behavior and extra functionalities. Here are a few examples:

- six set operations featuring union, intersection, difference, symmetrical difference, negation and concatenation.
- a hash cursor, computing a hash value for the recognized words along its path. This algorithm described in [DR92] realizes a bijective mapping between the recognized words of the DFA and integers providing a fast perfect hash function.
- default transition cursors using a variety of failure forward functions making for instance the automaton complete in a trivial way: if requested transition is a sink transition then the cursor moves along the default transition. Such cursors are used in the implementation of the Aho-Corasick pattern matching algorithm.

- string and C-string cursors giving to a simple word or sequence a cursor interface and therefore a flat automaton look.
- permutation cursor implements in a very simple way a virtual automaton recognizing all permutations of the word 123...n
- a scoring cursor computing an integer value for a matched pattern as the sum of all the scores of the matched sub-expressions. This was used in a search engine to evaluate the amount of confidence for each piece of matched text.

In the following three sections, we will take a closer look at two of these adapters, the set operations and the permutation cursor which highlight the cursor encapsulation power.

Set Operations. Union, intersection, difference, symmetrical difference and concatenation cursors are binary forward cursor adapters. They make use of two underlying forward cursor and provide the same complying interface. They perform on-the-fly all algorithmic operations needed and can be immediately used. The following piece of code check if word `word` is in the recognized language of the intersection of two DFAs `A1` and `A2` :

```
char word[] = "word to check";
DFA A1, A2;
forward_cursor<DFA> c1(A1.initial()), c2(A2.initial());
if (is_in(word, word + 13, intersection_cursor(c1, c2)))
  cout << "ok";
else
  cout << "not found";
```

The negation cursor is a unary forward cursor adapter on a DFA A allowing access to a virtual DFA whose language is $\Sigma^* \setminus L(A)$.

Permutations Automaton. This example shows a convenient way to hide data structure and to overcome memory space limitations and combinatorial explosion.
The regular expression matcher we developed had to look for n patterns combined in any possible order. The matcher engine had been previously written to search a range `[first,last)` of characters with a cursor `c` on the regular expression DFA:

```
bool match(Iterator first, Iterator last, ForwardCursor c) {
  for(; first != last && c.forward(*first); ++first)
     if (c.src_final()) return true;
  return false;
}
```

By simply changing the default cursor to a multiple cursor encapsulating n forward cursors we can look simultaneously for n patterns with the same algorithm. Assigning a unique integer $i \in [1, n]$ to each pattern, the problem comes

down to match a word in the DFA recognizing all permutations of the sequence $1, 2, 3, ..., n$.

Our first approach was to precompute the automaton for all permutations, but this quickly turns out to consume too much memory space when n reaches 9. The second step was to minimize the automaton which is very efficient: by using a compact data structure, we could shrink the automaton for $n = 8$ to about 20 Ko but the memory usage peak due to minimization and the pretty long processing time remained major drawbacks. Eventually, we created a cursor moving on a virtual automaton simply by keeping tracks of the letters of visited transitions in a bit vector. A vector full of 1 means you have reached an accepting state.

5 Stack Cursor

A stack cursor is in some sense a bidirectional cursor. By keeping track of its previous positions during iteration, it is able to move back.

5.1 Definition

A stack cursor is a forward cursor using a stack to store its path along the automaton allowing thus backward iteration. It is constituted of a stack of forward cursors and all operations apply to the stack top. Its behavior relies on the underlying forward cursor.

5.2 Interface

The interface of stack cursor is very close to the forward cursor interface: the `forward` action pushes the resulting cursor on to the top and the `backward` action pops it.

Remark that the comparison operator compares not only tops but entire stacks as the underlying concept is path and not transition. The reason will be made clear from the depth-first cursor abstraction in Sect. 6.

A stack cursor must implement the forward cursor requirements plus:

From the default stack cursor implementation we designed an adapter called the neighbor cursor.

5.3 Application: The Neighbor Cursor

This is the central issue of a spelling corrector design: given a word w and an editing distance d, find all words w' recognized by an automaton A for which the edition operations consisting of successive letter insertions, deletions and substitutions needed to reach w' from w do not exceed a score of d. Each operation type is assigned a relative score (a weight) and the final distance is the scores sum.

The neighbor cursor adapts the default stack cursor in two ways:

Name	Expression	Semantics
forward	`x.forward();`	move along current stack top transition and push reached transition (p, a', p')
forward with letter	`x.forward(a);`	move along transition labeled with `a` push $(q, a, \delta_1(q, a))$ and return `true` if $\delta_1(q, a) \neq 0$
backward	`x.backward();`	pop. return `false` if resulting stack is empty
comparison	`(x == y)`	return `true` if `x` stack is a copy of `y` stack

1. It has the responsibility to hide paths straying too far away from w.
2. It has to adapt the stacking policy to manage the deletion operation: when moving forward, it has to move on to the next letter of w but has to stay on the current transition. That means pushing a copy of the cursor stack top leaving it unchanged.

6 Depth-First Iteration Cursor

6.1 Definition

A depth-first cursor, as the name suggests accesses the transitions of a DFA in a depth-first order. It relies on the stack cursor implementation but belongs to a different concept: a depth-first cursor represents an algorithm stage rather than a mere position in the automaton.

6.2 Properties

1. A depth-first cursor iterates on transitions in a depth-first order.
2. A depth-first cursor never pushes sink transitions (the underlying stack holds only dereferenceable cursors). When reaching a sink transition, the action taken is conceptually equivalent to a `pop`.
3. A depth-first cursor represents an algorithm stage and consequently can be used to define algorithm applying ranges passed to a function.
4. It is default constructible, assignable, equality-comparable, dereferenceable, incrementable and by default represents the empty stack.

6.3 Interface

Name	Expression	Semantics
source	`x.src();`	return q
aim	`x.aim();`	return p
letter	`x.letter();`	return a
final source	`x.src_final();`	return `true` if $q \in F$
final aim	`x.aim_final();`	return `true` if $p \in F$
forward	`x.forward();`	move on to the next transition in depth-first order. return `true` if `x` actually moved forward or `false` if `x` poped.
comparison	`(x == y)`	compare entire stacks.

6.4 Algorithm Depth-First Iteration

To grant more freedom to algorithms users we will use depth-first cursors to define algorithm applying range. Starting from a position in the automaton given by a cursor set to a specified transition, algorithms will apply until a stop condition becomes true, most of the time until the stack is empty. This stop condition can be represented by a cursor too. Remember depth-first cursors are algorithm stages. Consequently, initializing a depth-first cursor with the empty stack makes it a valid end-of-range position and algorithm stops when the moving cursor reaches the empty stack state.

6.5 Application: The Language Algorithm

```
void language(DepthFirstCursor first, DepthFirstCursor last) {
  vector<char> word;
  while (first != last) {               // until end of range
    word.push_back(first.letter());     // push current letter
    if (first.aim_final()) output(word);// if target is final
                                        //    output the word
    while (! first.forward())           // while moving backward
      word.pop_back();                  // pop the last word
                                                  letter
  }
}
```

Example: display the automaton A language (Fig. 3)

```
DFA A;
// initialize start of range with state 1:
forward_cursor<DFA> begin(A.initial());
// set the cursor on transition (1,a,2):
begin.first_transition();
// use empty stack (default value) for end of range:
language(depth_first_cursor(begin), depth_first_cursor());
```

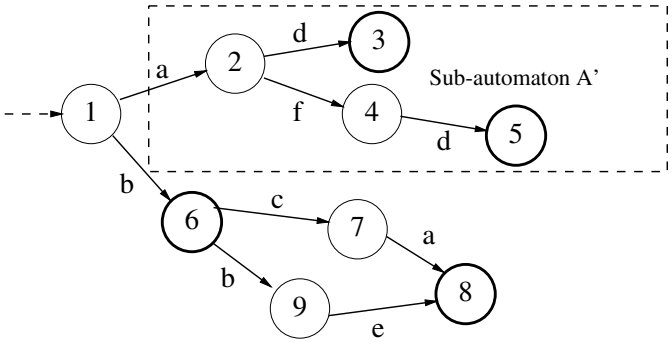


Fig. 3. Automaton A

Restricted range: display the sub-automaton A' language

```
// initialize start of range with transition (1,a,2):
forward_cursor<DFA> begin(A.initial()), end(A.initial());
begin.first_transition();
// initialize end of range with transition (1,b,6):
end.find('b');
language(depth_first_cursor(begin), depth_first_cursor(end));
```

7 Applying Algorithms

The main benefit of cursors is to allow to decide at the last minute in which way to apply an algorithm rather than at design time. Most of the time, one of the following policies is more adapted to one's need but the algorithm cannot be aware of it. Instead of writing three versions of the code, cursors allow external decisions.

7.1 On-the-Fly Processing

All cursors perform on-the-fly data processing. This has many advantages:

1. By writing one version of an algorithm you get the other two implementations for free (lazy and copy).
2. Sometimes combinatorial problems get in the way and there is no means to build the desired automaton but a cursor adapter is able to simulate a DFA (see Sect. 4.4 about permutations).
3. Sometimes operation is only punctual and there is no need to build an entire automaton as in Sect. 4.4 for intersection.

Consequently, algorithms impose only constraints on the public interface and possible extra internal processing is left up to the user.

7.2 Lazy Construction

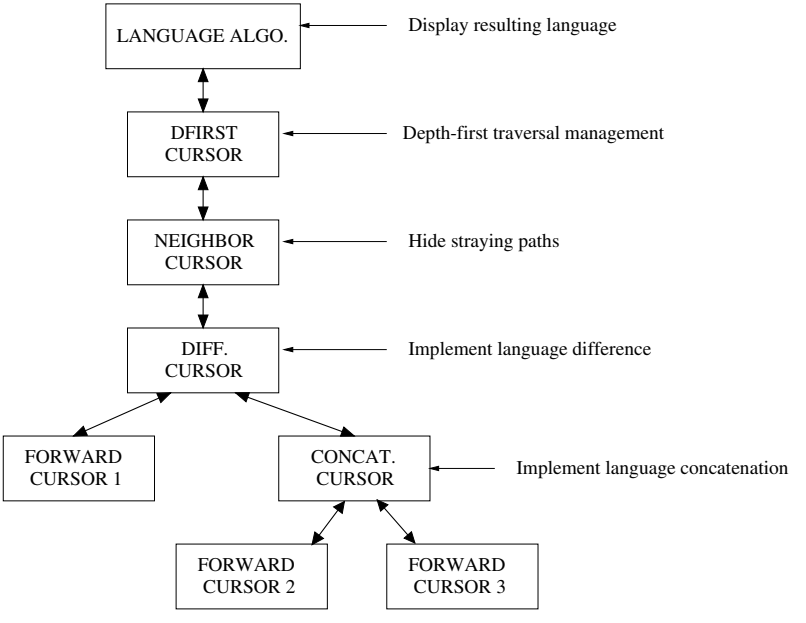
When the ratio of preprocessing time against computing time becomes too large, it is necessary to delay the actual construction or more precisely to incrementally build the resulting automaton. A famous example is the incremental construction of a DFA from a regular expression during the scanning of the text in [ASU86]. It is also particularly interesting in determinizing a NFA.
In this case, construction is made internally by a cursor adapter called the lazy cursor completely encapsulating the extra processing. This adapter is passed to the algorithm where a default cursor would be passed in an on-the-fly operation.

7.3 Building by Copy

Whenever the actual resulting DFA construction is needed, one can either use the `clone` algorithm which makes an exact copy from a cursors range into a new DFA or the `ccopy` algorithm (cursor copy) which duplicates the input DFA through a cursor in ascending stage of the depth-first iteration, trimming unneeded paths leading to non accepting states.

7.4 Algorithm Combining

Cursors offer a simple and straightforward way to extend algorithm power. For example, one can retrieve the language of the difference of two languages concatenation and the language of a DFA within an editing distance of 5 from a specified word:



By directly calling the `language` algorithm described in Sect. 6.5 we process the data on-the-fly but we can as well build the resulting automaton A by using the `ccopy` algorithm:

```
ccopy(A, depth_first_c(           // start of range
             neighbor_c(
               diff_c(forward_c(dfa1),
                      concat_c(forward_c(dfa2), forward_c(dfa3)),
               "word", 5))),
         depth_first_c());        // default end of range
```

8 Conclusion

Speed tests have been conducted to compare with classical recursive implementations speeds and it turned out that no time was waste because cursors are simple objects with very basic abilities: their implementation does not require a huge and complicated piece of code and most of the time a simple, optimal and straightforward solution does the trick.
An extended version of this document provides a rigorous detailed description of cursors behavior including time complexities and C++ signatures of methods [1]. Cursors have been successfully implemented in a search engine called word grep [2]. We have overcome most of the problems encountered in the first stage of ASTL development and encouraging results are leading us to consider extending them to cyclic automata and transducers.

References

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986, pp 157-165.

[DR92] D. Revuz. *Dictionnaires et Lexiques - Méthodes et Algorithmes.* PhD. thesis, université Paris VII, 1992, pp. 66-70.

[HU79] J. E. Hopcroft, J. D. Ullman. *Introduction to automata, languages and computation.* Addison-Wesley, 1979.

[MS89] D. R. Musser, A. Stepanov. *Generic Programming.* Lecture Notes in Computer Science 358, Springer-Verlag, 1989.

[SGI99] *Standard Template Library Programmer's Guide* Silicon Graphics Computer Systems, `http://www.sgi.com/Technology/STL/`, 1999

[SL95] A. Stepanov, M. Lee. *The Standard Template Library.* Hewlett-Packard laboratories, 1995.

[VLM98] Vincent Le Maout. *Tools to Implement Automata, a first step: ASTL.* Lecture Note in Computer Science 1436, Springer-Verlag, 1998, pp 104-108.

[1] ASTL and cursors v1.0 are freely available at `http://www-igm.univ-mlv.fr/~lemaout`

[2] Enhanced regular expressions defined on words

An Automaton Model of User-Controlled Navigation on the Web

K. Lodaya and R. Ramanujam

The Institute of Mathematical Sciences C.I.T. Campus, Chennai 600 113, India.
{kamal,jam}@imsc.ernet.in

Abstract. We present a simple theoretical model of web navigation, in which each WWW user creates *style specifications* which constrain web browsing, search and navigation using the user's own judgement of the quality of visited sites. A finite state automaton is associated with each specification, which is presented in a two-level modal logic making up the acceptance condition for the automaton. We show that many interesting queries regarding the user's web search can be answered using standard automata theory.

1 Motivation

A classical application of automata theory is searching a given text for occurrence of patterns. Finite state automata are employed in lexical analysers and document processors for pattern search. Recently, the world of documents has grown very fast, thanks to the World Wide Web (WWW), where we do not speak of text, but **hyper**text. Here documents contain pointers or *links* to documents elsewhere. However, access to documents is not only via these links but also using *search engines* which, given a keyword, locate a document containing the keyword. In such a situation, a natural application of automata theory seems to be that of web navigation.

On the other hand, we could argue that the distributed nature of information on the net is irrelevant and we can consider the entire web as one single huge piece of text. Search engines like Alta Vista or Google in effect do that, act as finite state automata that access the entire web when they search for keywords. In this manner, one can argue that there is no need for any new automata theory for searching the web, but only cleverer methods for data representation.

While there is some measure of truth in this argument, we believe that there is need for automata theory in this context from a slightly different viewpoint, that of quality controlled navigation. Quality remains the biggest challenge of search engines [Hen99], and there is a perceived urgent need for ways by which web users can direct and control navigation based on *their own judgement* of quality of web sites.

In this paper, we present a simple (naïve) theoretical model of user controlled navigation on the web. In this model, each user prepares a style sheet in which she specifies her judgement of sites based on the information they contain, and sets

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 208–216, 2001.

down constraints that every navigation must satisfy. The style sheet induces a finite state automaton that can be thought of as her own navigation-cum-search engine, which uses some general purpose engine like Alta Vista to access the web. These automata can be seen as customized front-ends to standard search engines. Formally, they are finite state transition systems, and their behaviour is given by partitioning the set of possible runs (or *trajectories*, as they are called here) into good and bad runs based on the constraints specified in the style sheet.

We then have a situation where the input to an automaton is given as a finite graph whose nodes carry input information, and the finite state control dictates which nodes to visit, in what order. It defines a process which originates from a home location, migrates to various nodes on the web picking up information on the way and returns home eventually.

The main contribution of the paper is a framework for the theoretical study of quality controlled web navigation. The framework is built on standard concepts and hence offers hope for implementation. It must be emphasized that the paper presents only the initial steps of a theoretical framework, and it is hoped that more realistic models will follow. However, design features of such models (like whether links should be static or dynamic, bookmarks implicit or explicit, search distributed or not, whether transitive closure of paths on the web is needed) should be decided by empirical compulsions rather than theoretical elegance. Corpus studies like [SHMM98] should dictate how this theory should proceed.

Related work: User-controlled navigation has been studied by several researchers from the WWW community, under the name of *spiders, crawlers, robots, knowbots* and so on. Miller and Bharat [MB98] give a kind of style specification using C++. Mendelzon, Mihaila and Milo [MMM96] modify the query language SQL to operate on the web. Pazzani, Muramatsu and Billsus [PMB96] describe a system which piggybacks onto a Lycos browser.

However, the use of automata theory for modelling these systems appears to be new. Automata working on graph structures have been studied for long: the important topic of visiting all sites in a labyrinth is surveyed in [KUK93]. Thomas [Tho97] has recently emphasized the importance of studying automata on partially ordered structures. All these papers concentrate on the class of graphs accepted/traversed by such automaton models; but, as we will see, the approach followed here is distinct. Even if we were to consider the Web as graph input to the automaton, due to the keyword search facility, the automaton is not obliged to follow the paths in the graph.

2 The Model

In this paper, automata behave in a somewhat non-standard fashion: there is one single finite graph processed by all automata. The behaviour of an automaton is some set of trajectories in this graph (as opposed to a set of graphs). Each node of the graph is supposed to contain some information, represented by a finite set of letters from a fixed alphabet. The finite state control moves on the

graph not linearly, but according to a specified constraint, either using an edge or using a *search*; in general the automaton does not process the entire input graph, but its behaviour is given by trajectories that take it from a designated home location back again to the home location visiting nodes as given by a *navigation constraint*. The input graph is meant to represent the internet, with URLs as nodes and links as the edge relation. Each automaton is thought of as a *search strategy* or a search engine defined by a user whose home location on the net defines the start node for the automaton. Every run of the automaton navigates the net in some way. The user controls this navigation by specifying requirements on the **quality** of information in accessed sites. Therefore, the (static) description of the net comes with a specification of information available at each node, and each user automaton includes a quality judgement of sites, based on information they contain. We use letters of a finite **information alphabet** to abstractly associate keywords with each node, and a set of propositional letters to denote quality judgement.

Which locations in the net are accessible to any specific user ? In principle, all. However, if the user can access arbitrary locations only by using a search engine, then the vocabulary used to query the engine effectively limits the range of locations accessed. This is reflected in the model below.

2.1 Navigators

A **web** is a tuple $W = (U, L, I, \iota)$, where U is a finite set of locations (URLs), $L \subseteq (U \times U)$ is the link relation, I is a finite information alphabet, $\iota : U \to 2^I$ is the content map.

We use $u, v, \ldots$ to refer to elements of U and $a, b, \ldots$ to refer to elements of I. Below, we will define a logical language in which we can talk about specific locations, connectivity between locations and navigation paths on the net. Let $P = \{p_0, p_1, \ldots\}$ be a countable set of propositional letters. The set of all formulas of the logic is denoted Φ, and we use ϕ, ϕ' etc to refer to formulas in Φ. P_ϕ denotes the set of propositions occurring in ϕ. For the rest of this section, fix a web $W = (U, L, I, \iota)$.

A **navigator** on W is a tuple $N_W = (h, Voc, \chi, \phi)$, where $h \in U$; Voc, the **vocabulary** of N_W, is a nonempty subset of I such that $\iota(h) \subseteq Voc$; $\chi : 2^{Voc} \to 2^{P_\phi}$ is a quality map and $\phi \in \Phi$. $U(N_W) \stackrel{\text{def}}{=} \{u \in U \mid (\iota(u) \cap Voc) \neq \emptyset\}$, is said to be the set of locations visible to N_W. $Val(N_W) : U(N_W) \to 2^{P_\phi}$ is the induced valuation defined by: $Val(N_W)(u) = \chi(\iota(u) \cap Voc)$.

h is the home location of the navigator N_W. Note that h is always visible to N_W. χ is a map that indirectly assigns judgements to locations in U, which is given by the induced valuation map.

Let $N_W = (h, Voc, \chi, \phi)$ be a navigator on W. The **subnet visible to** N_W is given by: $W' = (U', L', Voc, \iota')$, where $U' = U(N_W)$, $L' = L \cap (U' \times U')$ and $\iota' : U' \to 2^{Voc}$ is defined by: $\iota'(u) = \iota(u) \cap Voc$. W' is denoted as $W\lceil N_W$.

A **trajectory** of N_W is a *finite* sequence $\rho \in (U(N_W))^*$ of the form $u_0 u_1 \cdots u_K$, $K \geq 0$, where $u_0 = u_K = h$. Thus a trajectory is an itinerary

of locations visited on the net. We do not explicitly represent bookmarking visited locations; as we will see below, the navigation constraint ϕ can refer to sites that have been visited already, and this provides an implicit way of referring to bookmarks in trajectories.

Let $Tr(N_W)$ denote the set of all trajectories of N_W. The **behaviour** of N_W is given by the set $Beh(N_W) \stackrel{\text{def}}{=} \{\rho \in Tr(N_W) \mid \rho \models \phi\}$, where the relation $\rho \models \phi$ is defined below. Thus, we need to give the syntax and semantics of the logic to complete the definition of the model.

The formulas of the logic are presented in two layers. The lower layer consists of **location formulas**, and their syntax is given as follows:

$$\Gamma ::= p \ \in \ P \mid \neg \ \alpha \mid \alpha_1 \ \vee \ \alpha_2 \mid \langle from \rangle \ \alpha \mid \langle to \rangle \alpha$$

The other logical connectives $\wedge, \supset, \equiv$ are defined as usual. The dual modalities are given by:
$[from]\alpha \stackrel{\text{def}}{=} \neg\langle from\rangle\neg\alpha$ and $[to]\alpha \stackrel{\text{def}}{=} \neg\langle to\rangle\neg\alpha$.

The syntax of **navigation formulas** is given as follows, where $\alpha \in \Gamma$:

$$\Phi ::= \alpha \mid \ \neg\phi \mid \phi_1 \ \vee \ \phi_2 \mid \bigtriangledown\alpha \mid \triangleright \ \phi \mid \alpha?\phi \mid \Diamond\phi$$

Location formulas specify how locations with specific quality of information are interlinked in the net, with $\langle from\rangle\alpha$ asserting the existence of a link from a node satisfying α to the current node, and $\langle to\rangle$ asserting the other way about. $\bigtriangledown\alpha$ refers to an implicit bookmark to a site where α holds; it says that such a site has been visited in the trajectory at this stage, and hence that such information is available to the user. $\triangleright\phi$ asserts that the navigation given by ϕ continues along one of the links available at the current location (by a 'click'), whereas $\alpha?\phi$ directs a *search* for a location satisfying α, from where the navigation ϕ continues. $\Diamond$ represents eventuality, and its dual is given by $\Box\phi \stackrel{\text{def}}{=} \neg\Diamond\neg\phi$.

Let $N_W = (h, Voc, \chi, \phi)$ be a navigator, $W' = (U', L', Voc, \iota')$ the subnet of W visible to N_W and $Val(N_W)$ the valuation induced by N_W. Location formulas are interpreted over locations of W'. $W', u \models_l \alpha$ denotes that α holds in location u in net W'.

- $W', u \models_l p$ iff $p \in Val(N_W)(u)$.
- $W', u \models_l \neg\alpha$ iff $W', u \not\models_l \alpha$.
- $W', u \models_l \alpha \vee \beta$ iff $W', u \models_l \alpha$ or $W', u \models_l \beta$.
- $W', u \models_l \langle from\rangle\alpha$ iff there exists u' such that $(u', u) \in L'$ and $W', u' \models_l \alpha$.
- $W', u \models_l \langle to\rangle\alpha$ iff there exists u' such that $(u, u') \in L'$ and $W', u' \models_l \alpha$.

Let $\rho = u_0 u_1 \cdots u_K \in Tr(N_W)$. For $\phi \in \Phi$ and $k \in \{0, \ldots, K\}$, the notion $\rho, k \models \phi$ is again defined inductively.

- $\rho, k \models \alpha$, for $\alpha \in \Gamma$, iff $W', u_k \models_l \alpha$.
- $\rho, k \models \neg\phi$ iff $\rho, k \not\models \phi$.
- $\rho, k \models \phi_1 \vee \phi_2$ iff $\rho, k \models \phi_1$ or $\rho, k \models \phi_2$.

- $\rho, k \models \bigtriangledown\alpha$ iff there exists $m : 0 \leq m \leq k : W', u_m \models_l \alpha$.
- $\rho, k \models \triangleright\phi$ iff $k < K$, $(u_k, u_{k+1}) \in L'$ and $\rho, k+1 \models \phi$.
- $\rho, k \models \alpha?\phi$ iff $k < K$, $W', u_{k+1} \models_l \alpha$ and $\rho, k+1 \models \phi$.
- $\rho, k \models \Diamond\phi$ iff there exists $m : k \leq m \leq K$ such that $\rho, m \models \phi$.

We use the notation $\rho \models \phi$ when $\rho, 0 \models \phi$. This completes the definition of navigator N_W. Note that formulas do not directly refer to the information contained in locations, but only to the quality of such information, as given by χ. $\alpha?\phi$ is more natural here, but we could have formulas $w?\phi$, where w is a keyword in Voc, without affecting our results.

3 Analysis

Given a web W and a navigator N_W on it, we can analyse W' and N_W to answer a number of queries regarding the 'search engine' given by N_W. The following queries seem natural: Does the navigation meet some basic quality requirement ? Is a site with some specific information visited at all ? Is it the case that every visited site of some quality has links only to sites having the same quality of information ? Is the navigation constraint consistent ? That is, is $Beh(N_W) \neq \emptyset$? Many of these and other related questions can be answered by a simple technique [VW86]: construct a verifying automaton (an ordinary NFA) which captures the behaviour of the given navigator, and run standard algorithms on that automaton. Below we describe how we can associate such an automaton with every navigator.

Define as usual the subformula closure of any formula $\phi \in \Phi$, denoted $CL(\phi)$. The size of $CL(\phi)$ is linear in the length of ϕ. Let $\triangleright_N$ denote the set of all formulas in $CL(\phi)$ of the form $\triangleright\psi$. Similarly let $?_N$ contain all $\alpha?\psi$ formulas in $CL(\phi)$. Let $Nxt \stackrel{\text{def}}{=} \triangleright_N \cup ?_N$.

Let $A \subseteq CL(\phi)$. A is said to be an atom iff it satisfies the following conditions:

- For all $\neg\psi \in CL(\phi)$, $\neg\psi \in A$ iff $\psi \notin A$.
- For all $\phi_1 \vee \phi_2 \in CL(\phi)$, $\phi_1 \vee \phi_2 \in A$ iff $\phi_1 \in A$ or $\phi_2 \in A$.
- $A \cap ?_N = \emptyset$ or $A \cap \triangleright_N = \emptyset$.
- If $A \cap Nxt = \emptyset$ then for all $\Diamond\psi \in A$, $\psi \in A$.

Assume that we are given a web $W = (U, L, I, \iota)$, and navigator $N_W = (h, Voc, \chi, \phi_0)$ on it. χ would be presented as a list of boolean formulas on Voc, one for each $p \in P_{\phi_0}$.

Let $|\phi_0| = m_0$. Let CL denote the subformula closure of ϕ_0. Fix an ordering $\preceq$ of CL such that if $|\phi_1| \leq |\phi_2|$ then $\phi_1 \preceq \phi_2$. Let $CL(i)$ denote the i^{th} formula in the enumeration, and let all formulas from Γ precede those from $\Phi - \Gamma$.

Step 1: Construct $W \lceil N_W = (U', L', Voc, \iota')$. This construction, which takes $O(|U|)$ time in the worst case, depends crucially on how ι is represented. Let $M = |U'|$. Fix an enumeration of locations in U'.

Step 2: Construct a $|(CL \cap \Gamma)| \times M$ boolean array μ such that $\mu(i, j) = 1$ iff $W', u \models_l \alpha$, where $CL(i) = \alpha$ and u is the j^{th} location in the enumeration of

W'. (We will use the notation $\mu(\alpha, u) = 1$ in this case.) This construction can be done in $O(M.m_0)$ time. However, we can identify locations as follows: $u_1 \sim u_2$ iff for all $\alpha \in (CL \cap \Gamma)$, $\mu(\alpha, u_1) = \mu(\alpha, u_2)$. Thus we only need a boolean $m_0 \times 2^{m_0}$ array. Let $\mu(\alpha, [u])$ represent these elements.

Step 3: Let AT denote the set of all subsets of CL which are atoms. Let $Vis \stackrel{\text{def}}{=} \{\alpha \in \Gamma \mid \bigtriangledown\alpha \in CL(\phi)\}$. The states of our verifying automaton $\mathcal{A}_N \stackrel{\text{def}}{=} (Q, \Rightarrow, I, F)$ keep track of these formulas in addition to the URLs visited:

- $Q = \{([u], A, S) \mid u \in L', A \in AT, S \subseteq Vis$ and for all $\alpha \in \Gamma$: $\alpha \in A$ iff $\mu(\alpha, [u]) = 1$ and if $\bigtriangledown\alpha \in A$ then $\alpha \in S\}$.
- $I = \{([h], A, A \cap Vis) \mid A \in AT, \phi_0 \in A\}$.
- $F = \{([h], A, S) \mid A \in AT$ such that $A \cap Nxt = \emptyset\}$.
- $([u], A, S) \Rightarrow ([v], B, S')$ iff
  1. $A \cap Nxt \neq \emptyset$,
  2. if $\triangleright\phi \in A$ then $\phi \in B$ and $([u], [v]) \in [L']$,
  3. if $\alpha?\phi \in A$ then $\mu(\alpha, [v]) = 1$ and $\phi \in B$,
  4. if $\Diamond\phi \in A$ and $\phi \notin A$ then $\Diamond\phi \in B$, and
  5. $S' = S \cup (B \cap Vis)$.

For $q = ([u], A, S) \in Q$, we use the notation $q\lceil U$ to denote u. Let $\rho = u_0 u_1 \ldots u_k \in U^*$. We say that ρ is **accepted** by A_N iff there exists a sequence $q_0 q_1 \ldots q_k$ such that $q_0 \in I, q_k \in F$, and for all $i : 0 \leq i < k$, $q_i \Rightarrow q_{i+1}$ and $\rho = q_0\lceil U \ldots q_k\lceil U$. Let $Lang(\mathcal{A}_N)$ denote the set of all strings in U^* accepted by $\mathcal{A}_N$.

Theorem 1. *If $\mathcal{A}_N$ is the automaton associated with navigator N_W on web W, then $Beh(N_W) = Lang(\mathcal{A}_N)$.*

The construction takes $O(|U|) + O(M.m_0) + 2^{c.m_0}$, that is, $O(|U|) + 2^{c.m_0}$ time, where c is a (small) constant. We can then answer queries like the ones listed in the beginning of this section. Since the number of URLs in the web is large, the fact that the $O(|U|)$ factor is additive is crucial. It can then be seen as a preprocessing step for the algorithm. $M = |U'|$ is usually much smaller, and hence the multiplicative factor is more acceptable.

After elimination of useless states and transitions, we can have an efficient representation of $\mathcal{A}_N$ that helps us do reachability analysis, enumerate connected components, and so on. Then, checking whether a location is visited, or whether there exists a trajectory visiting all locations in a given set, is simple.

4 A Web Example

An example researcher who looks for grants dealing with work on genomes is considered by Pazzani, Muramatsu and Billsus [PMB96]. We consider a style sheet for such a researcher. The web $W = (U, L, I, \iota)$ is given, and a relevant portion

of the structure (U, L, ι) is shown in Figure 1. The style sheet is given by $N_W = (\texttt{Home}, Voc, \chi, \phi)$, where $Voc = \{grant, genome, institute, foundation, project, experiment, cloning, publist, appform\}$, and $I = Voc \cup \{chromosome\text{-}22, sex, \ldots\}$.

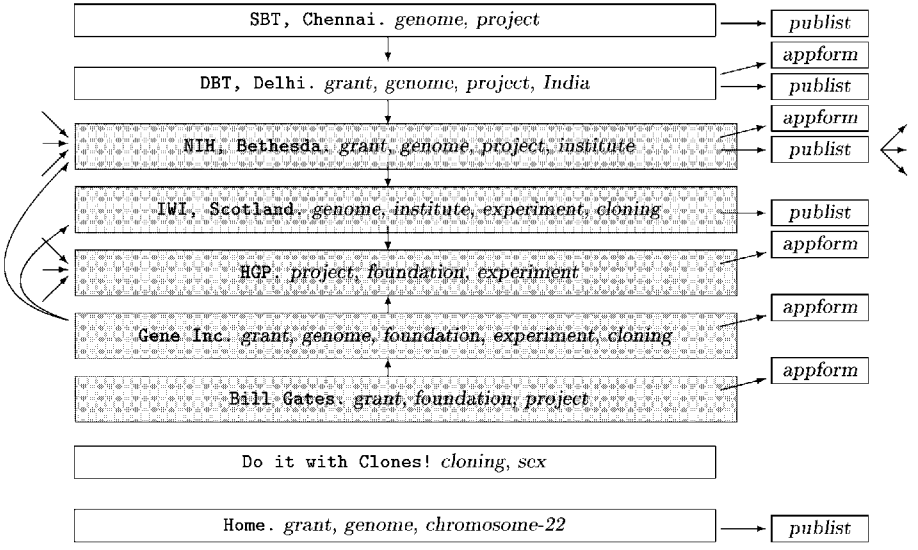


Fig. 1. Grants and genomes on the web

The set of propositions consists of $\{commercial, government, authority, hub, spam, submit, interesting, highstatus, money, \ldots\}$. For our example, we assume the only commercial site is `Gene Inc` and the only government site is `DBT`. *Authority* and *hub* [Hen99] refer to a URL having high in- and out-degree respectively. These are indicated in the figure by a few arrows coming into and going out of a node, respectively. *Spam*, *submit*, *interesting* are true in URLs which have *sex*, *appform* and $\{grant, genome\}$ respectively.

The two propositions listed last decide parameters of interest to our researcher: whether the URL represents a place which commands status and whether it has money. Sites satisfying *highstatus* are shown shaded in the figure: the proposition is true in those sites which represent a foundation, or which do research in cloning, or which have a large list of publications (that is, a link to a site with keyword *publist* which satisfies *hub*). *Money* is true in the two government and commercial sites, as well as in all the sites having the keyword *foundation*.

This completes the (indirect) definition of the quality function. We now come to the navigation part. The navigation constraint ϕ is defined as the conjunction of formulas $\phi_i, i \in \{1, \ldots, 5\}$ given below.

$\phi_1 = \Box\neg spam$ eliminates junk sites.
$\phi_2 = \Box(\neg highstatus \supset \bigtriangledown highstatus)$ ensures that a *highstatus* site is visited first.
$\phi_3 = \Box(authority \supset \bigtriangledown\neg authority)$ similarly prioritizes sites which are less well known (and presumably have fewer contenders for grants).
$\phi_4 = \Box(\langle to\rangle submit \supset \triangleright submit)$ visits a link containing an application form.
$\phi_5 = interesting \wedge money?\Diamond(interesting \wedge money?\Diamond(interesting \wedge money))$ visits three sites, according to the user's rating and priorities, which are *interesting* and have *money*.

Having constructed the automaton, the user can now ask questions of interest: Is `Gene Inc` on a particular trajectory? Is there a trajectory visiting all interesting sites with money? and so on.

5 Discussion

The framework presented here is intended only as preliminary. As mentioned in Section 1, the design of the logical language should be guided more by application requirements than theoretical considerations. However, some simple extensions can be made without drastically altering the current framework.

- $\rho, k \models \phi_1 \mathbf{B} \phi_2$ iff if there exists $m \geq k$ such that $\rho, m \models \phi_2$, then there exists $l : k \leq l < m$ such that $\rho, l \models \phi_1$. (This allows priority in visiting locations.)
- $\rho, k \models \triangleleft\phi$ iff $0 < k < K$, $u_{k-1} = u_{k+1}$ and $\rho, k+1 \models \phi$. (Back button.)
- $\rho, k \models \trianglerighteq\phi$ iff there exists $m \geq 0$ such that $k + m \leq K$, $\{u \mid (u_k, u) \in L\} = \{u_{k+1}, \ldots, u_{k+m}\}$ and and $\rho, k + m \models \phi$. (Visit all local edges.)
- $\rho, k \models \mathbf{E}\alpha$, iff $\{u \mid W', u \models_l \alpha\} \subseteq \{u_0, u_1, \ldots u_k\}$. (Exhaustive search.)
- $\rho, k \models \alpha??\phi$, iff $k < K$, $W', u_{k+1} \models_l \alpha$, $u_{k+1} \notin \{u_0, \ldots, u_k\}$ and $\rho, k+1 \models \phi$. (New location.)

The first two specifications can be easily incorporated into the automaton construction without increasing the complexity. The third case is rather tricky: the surprise is that it can be carried out without increasing the complexity. For the last two modalities, the automaton construction is easy to modify, but the cost increases dramatically. In the states of $\mathcal{A}_N$, we need to carry the set of u-equivalence classes visited, so that the check for exhaustive visit or new visit can be made. Because of this, the construction becomes doubly exponential in the size of the navigation constraint in N_W. However, if we were interested only in the construction of $\mathcal{A}_N$ with the property that $Beh(N_W) \neq \emptyset$ iff $Lang(\mathcal{A}_N) \neq \emptyset$, this could be done in singly exponential time.

There are other interesting and desirable extensions which are not minor and require reworking the theory. For instance, there is a good reason to want quantification over locations, or over information objects. Extending the framework to include page structure within URLs is easy though rather messy; a different formulation may be more convenient.

Another direction is explicit representation of *actions* in the framework, like bookmarking locations, downloading specific information objects, etc. An even

more challenging step is to study dynamic nets, where links are made and broken during the course of navigation. This would lead to the consideration of several navigators cruising the web simultaneously, perhaps exchanging information among themselves.

Such a picture raises the obvious question of **information security** which is not modelled here at all. The style sheets envisioned here will be realistic only if security considerations become part of navigation specifications.

References

[Hen99] M. Henzinger: Tutorial on Web Algorithms, joint session of *FST & TCS* and *ISAAC*, Chennai (1999).

[KUK93] V.B. Kudryavstsev, Sh. Ushchumlich and G. Kilibarda: "On behaviour of automata in labyrinths", *Discrete Math. Appl.* **3**,1 (1993) 1–28.

[MMM96] A.O. Mendelzon, G.A. Mihaila and T. Milo: "Querying the world wide web", in *Symposium on Parallel & Distributed Information Systems* (1996).

[MB98] R.C. Miller and K. Bharat: "Sphinx: a framework for creating personal, site-specific web crawlers", in *Proceedings of the 7th International World Wide Web Conference*, Brisbane (1998).

[PMB96] M. Pazzani, J. Muramatsu and D. Billsus: "Syskill & Webert: identifying interesting web sites", in *Proceedings of the 13th National Conference on AI*, Portland (1996). Revised version in *Machine Learning* **27** (1997).

[SHMM98] C. Silverstein, M. Henzinger, H. Marais and M. Moricz: "Analysis of a very large Alta Vista query log", *SRC Technical Note* **1998-014**, `http://www.research.digital.com/SRC/`.

[Tho97] W. Thomas: "Automata theory on trees and partial orders", in *Proceedings of TAPSOFT*, *LNCS* **1214** (1997) 20–38.

[VW86] M.Y. Vardi and P. Wolper: "An automata-theoretic approach to automatic program verification", *Proceedings of the 1st LICS Conference*, Boston (1986) 332–334.

Direct Construction of Minimal Acyclic Subsequential Transducers

Stoyan Mihov[1] and Denis Maurel[2]

[1] Linguistic Modelling Laboratory
LPDP – Bulgarian Academy of Sciences
stoyan@lml.bas.bg
[2] LI (Computer Laboratory) – Université de Tours
E3i, 64 avenue Jean-Portalis, F37200 Tours, France
maurel@univ-tours.fr

Abstract. This paper presents an algorithm for direct building of minimal acyclic subsequential transducer, which represents a finite relation given as a sorted list of words with their outputs. The algorithm constructs the minimal transducer directly – without constructing intermediate tree-like or pseudo-minimal transducers. In NLP applications our algorithm provides significantly better efficiency than the other algorithms building minimal transducer for large-scale natural language dictionaries. Some experimental comparisons are presented at the end of the paper.

1 Introduction

For the application of large-scale dictionaries two major problems have to be solved: fast lookup speed and compact representation. Using automata we can achieve fast lookup by determinization and compact representation by minimization. For providing information for the recognized words we have to construct automata with outputs or transducers. The use of automata with labels on the final states for representation of dictionaries is presented by Dominique Revuz in [8]. In [6,7] Mehryar Mohri reviews the application of transducers for Natural Language Processing. He compares the benefits using subsequential transducers. The transducers are more compact in some cases and can be combined by composition or other relational operators. The transducers can be applied also for the reverse direction – to find the input words which are mapped to a given output.

In this paper we focus on building the minimal subsequential transducer for a given input list of words with their outputs. This is the procedure required for the initial construction of the transducer representing a dictionary. Earlier presented methods are building temporary transducers for the input list first, and later they have to be minimized. This temporary transducers can be huge compared to the resulting minimized one. For example in [6] Mehryar Mohri writes:

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 217–229, 2001.

> "But, as with automata, one cannot construct directly the p-subsequential transducer representing a large-scale dictionary. The tree construction mentioned above leads indeed to a blow up for a large number of entries. So, here again, one needs first to split the dictionary into several parts, construct the corresponding p-subsequential transducers, minimize them, and then perform the union of these transducers and reminimize the resulting one."

In [3] Denis Maurel is building efficiently the pseudo-minimal subsequential transducer, which can be significantly smaller than the tree-like transducer. The pseudo-minimal transducer has to be additionally minimized. Our experiments are showing that for large-scale dictionaries the pseudo-minimal subsequential transducer is about 10 times larger than the minimized transducer.

In this paper we present an algorithm for building minimal subsequential transducer for a given sorted list without the necessity of building any intermediate non-minimal transducers. The algorithm is a combination of the algorithm for direct construction of minimal acyclic Finite-State automaton given in [2,4] with the methods for construction of minimal subsequential transducers given in [3,5]. The resulting subsequential transducer is minimal.

In comparison with the approach of Mehryar Mohri we don't build minimal intermediate transducers for parts of the dictionary which after deterministic union have to be minimized again. We are proceeding incrementally word by word building the minimal except for the last word transducer.

2 Mathematical Concepts

In this section we present shortly the mathematical basics used in the algorithm for direct construction of minimal subsequential transducers. A more detailed presentation with the corresponding proofs for the minimal except for a word automata is given in [4].

2.1 Subsequential Transducers

Definition 1. *A p-subsequential transducer is a tuple* $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$, *where:*

- Σ *is a finite input alphabet;*
- Δ *is a finite output alphabet;*
- S *is a finite set of states;*
- $s \in S$ *is the starting state;*
- $F \subseteq S$ *is the set of final states;*
- $\mu : S \times \Sigma \to S$ *is a partial function called the transition function;*
- $\lambda : S \times \Sigma \to \Delta^*$ *is a partial function called the output function;*
- $\Psi : F \to 2^{\Delta^*}$ *is the final function. We will require that* $\forall r \in F \ (|\Psi(r)| \leq p)$.

The function μ is extended naturally over $S \times \Sigma^*$ as in the case for finite state automata:

$$\forall r \in S\ (\mu^*(r, \varepsilon) = r)\ ;\ \forall r \in S\ \forall \sigma \in \Sigma^*\ \forall a \in \Sigma\ (\mu^*(r, \sigma a) = \mu(\mu^*(r, \sigma), a)).$$

The function λ is extended over $S \times \Sigma^*$ by the following definition:

$$\forall r \in S\ (\lambda^*(r, \varepsilon) = \varepsilon);\ \forall r \in S \forall \sigma \in \Sigma^* \forall a \in \Sigma\ (\lambda^*(r, \sigma a) = \lambda^*(r, \sigma)\lambda(\mu^*(r, \sigma), a)).$$

The set $L(\mathcal{T}) = \{\sigma \in \Sigma^* \mid \mu^*(s, \sigma) \in F\}$ is called the input language of the transducer $\mathcal{T}$. The subsequential transducer maps each word from the input language to a set of at most p output words. The output function $O_{\mathcal{T}} : L(\mathcal{T}) \to 2^{\Delta^*}$ of the transducer is defined as follows:

$$\forall \sigma \in L(\mathcal{T})\ (O_{\mathcal{T}}(\sigma) = \lambda^*(s, \sigma) \cdot \Psi(\mu^*(s, \sigma))).$$

Two transducers $\mathcal{T}$ and $\mathcal{T}'$ are called equivalent when $L(\mathcal{T}) = L(\mathcal{T}')$ and $O_{\mathcal{T}} = O_{\mathcal{T}'}$.

Definition 2. *Let $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ be a subsequential transducer.*

1. *The state $r \in S$ is called reachable from $t \in S$, when $\exists \sigma \in \Sigma^*\ (\mu^*(t, \sigma) = r)$.*
2. *We define the subtransducer starting in $s' \in S$ as:*
   $\mathcal{T}|_{s'} = \langle \Sigma, \Delta, S', s', F \cap S', \mu|_{S' \times \Sigma}, \lambda|_{S' \times \Sigma}, \Psi|_{F \cap S'} \rangle$, where:
   $S' = \{r \in S \mid r$ is reachable from $s'\}$.
3. *Two states $s_1, s_2 \in S$ are called equivalent, when $\mathcal{A}|_{s_1}$ and $\mathcal{A}|_{s_2}$ are equivalent (when $L(\mathcal{T}|_{s_1}) = L(\mathcal{T}|_{s_2})$ and $O_{\mathcal{T}|_{s_1}} = O_{\mathcal{T}|_{s_2}}$).*

We cannot use directly the minimization algorithms developed for automata because in some cases by moving the output labels (or parts of them) along the paths we can get a smaller transducer. To avoid this we have to use transducers which has the property that the output is pushed back toward the initial state as far as possible. Mehryar Mohri [5] shows that there is a minimal transducer which satisfies this property. We will define this more formally bellow.

With $u \wedge v$ we denote the longest common prefix of the words u and v from Σ^* and with $u^{-1}(uv)$ we denote the word v – the quotient of the left division of uv by u. For the set of words $A = \{a_1, a_2, \ldots, a_n\}$ with $\bigwedge A$ we will denote the word $\bigwedge A = a_1 \wedge a_2 \wedge \ldots \wedge a_n$.

With $D(\mathcal{T})$ we will denote the set of the prefixes of $L(\mathcal{T})$:

$$D(\mathcal{T}) = \{u \in \Sigma^* \mid \exists w \in \Sigma^*\ (uw \in L(\mathcal{T}))\}.$$

We define the function $g_{\mathcal{T}} : D(\mathcal{T}) \to \Delta^*$ as follows:

$$g_{\mathcal{T}}(\varepsilon) = \varepsilon\ \ ;\ \ g_{\mathcal{T}}(u) = \bigwedge_{w \in \Sigma^*\ \&\ uw \in L(\mathcal{T})} \bigwedge O_{\mathcal{T}}(uw), \text{ for } u \in D(\mathcal{T}), u \neq \varepsilon.$$

Now we are ready to define the canonical subsequential transducer.

Definition 3. *The subsequential transducer* $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ *is called canonical if the following condition holds:*

$$\forall r \in S\ \forall a \in \Sigma\ \forall \sigma \in \Sigma^*\ ((\mu^*(s,\sigma) = r\ \&\ !\mu(r,a)) \rightarrow \lambda(r,a) = [g_{\mathcal{T}}(\sigma)]^{-1} g_{\mathcal{T}}(\sigma a))$$

We can see that the condition above corresponds to the property that the output is pushed back to the initial state as much as possible.

The subsequential transducer $\mathcal{T}$ is called minimal if any other transducer equivalent to $\mathcal{T}$ has more or equally many states as $\mathcal{T}$.

Theorem 1. *For every subsequential transducer* $\mathcal{T}$ *there exists a minimal canonical subsequential transducer equivalent to* $\mathcal{T}$.

Theorem 2. *If there are no different equivalent states in the canonical subsequential transducer* $\mathcal{T}$ *then* $\mathcal{T}$ *is minimal.*

A more complete presentation of the minimal subsequential transducers can be find in [5]).

2.2 Minimal Except for a Word Subsequential Transducer

Definition 4. *Let* $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ *be a subsequential transducer with input language* $L(\mathcal{T})$. *Then the transducer* $\mathcal{T}$ *is called* minimal except for the word $\omega \in \Sigma^*$, *when the following conditions hold:*

1. *Every state is reachable from the starting state and from every state a final state is reachable;*
2. ω *is a prefix of the last word in the lexicographical order of* $L(\mathcal{T})$;
   In that case we can introduce the following notations:

$$\omega = w_1^{\mathcal{T}} w_2^{\mathcal{T}} \dots w_k^{\mathcal{T}}\ ,\ \text{where } w_i^{\mathcal{T}} \in \Sigma\ ,\ \text{for } i = 1, 2, \dots, k \quad (1)$$

$$t_0^{\mathcal{T}} = s\ ;\ t_1^{\mathcal{T}} = \mu(t_0^{\mathcal{T}}, w_1^{\mathcal{T}})\ ;\ t_2^{\mathcal{T}} = \mu(t_1^{\mathcal{T}}, w_2^{\mathcal{T}})\ ;\ \dots\ ;\ t_k^{\mathcal{T}} = \mu(t_{k-1}^{\mathcal{T}}, w_k^{\mathcal{T}}) \quad (2)$$

$$T = \{t_0^{\mathcal{T}}, t_1^{\mathcal{T}}, \dots, t_k^{\mathcal{T}}\} \quad (3)$$

3. *In the set* $S \setminus T$ *there are no different equivalent states;*
4. $\forall r \in S\ \forall i \in \{1, 2, \dots, k\}\ \forall a \in \Sigma\ (\mu(r,a) = t_i \leftrightarrow (i > 0\ \&\ r = t_{i-1}\ \&\ a = w_i^{\mathcal{T}}))$;
5. $\mathcal{T}$ *is a canonical subsequential transducer.*

Example 1. An acyclic 2-subsequential transducer over the input alphabet $\{a, b, c\}$ is given on Figure 1. The input language of the transducer is [{apr, aug, dec, feb, jan, jul}.] The output function of the transducer is: $O(apr) = \{30\}; O(aug) = \{31\}; O(dec) = \{31\}; O(feb) = \{28, 29\}; O(jan) = \{31\}; O(jul) = \{31\}$. This transducer is minimal except for the word *jul*.

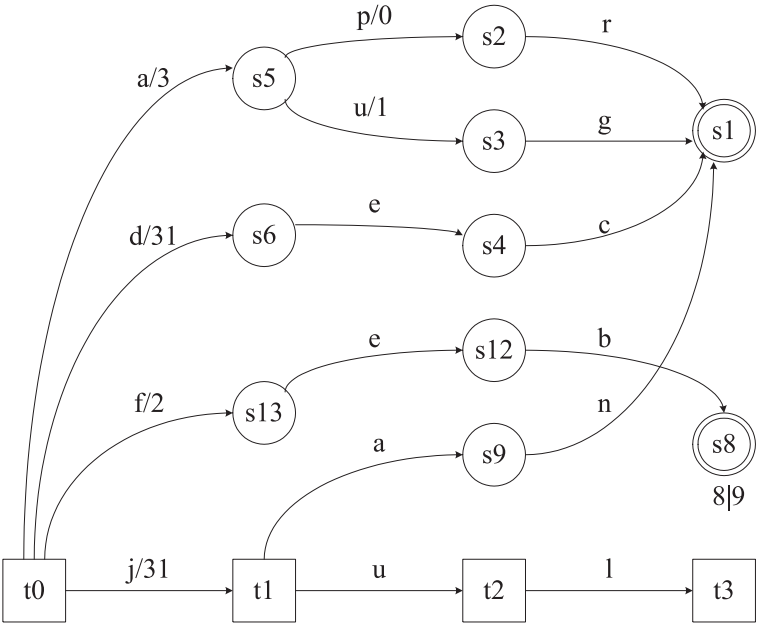


Fig. 1. Subsequential transducer minimal except for *jul*.

Proposition 1. *A subsequential transducer which is minimal except for the empty word ε is minimal.*

Lemma 1. *Let the subsequential transducer $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ be minimal except for $\omega = w_1 w_2 \dots w_k$, $\omega \neq \varepsilon$. Let there be no state equivalent to t_k in the set $S \setminus T$. Then $\mathcal{T}$ is also minimal except for the word $\omega' = w_1 w_2 \dots w_{k-1}$.*

Lemma 2. *Let the subsequential transducer $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ be minimal except for $\omega = w_1 w_2 \dots w_k$, $\omega \neq \varepsilon$. Let the state $p \in S \setminus T$ be equivalent to the state t_k. Then the transducer:*
$\mathcal{T}' = \left\langle \Sigma, \Delta, S \setminus \{t_k\}, s, F \setminus \{t_k\}, \mu', \lambda|_{S\setminus\{t_k\}\times\Sigma}, \Psi|_{S\setminus\{t_k\}} \right\rangle$ where:

$$\mu'(r,a) = \begin{cases} \mu(r,a) & \text{, in case } r \neq t_{k-1} \vee a \neq w_k \text{ and } \mu(r,a) \text{ is defined} \\ p & \text{, in case } r = t_{k-1}, a = w_k \\ \text{not defined otherwise} \end{cases}$$

is equivalent to the transducer $\mathcal{T}$ and is minimal except for the word $\omega' = w_1 w_2 \dots w_{k-1}$.

Lemma 3. *Let the subsequential transducer* $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ *be minimal except for* $\omega = w_1 w_2 \dots w_k$. *Then for* t_k *holds the following statement:*

$$t_k \text{ is equivalent to } r \in S \setminus T \leftrightarrow$$
$$((t_k \in F \leftrightarrow r \in F) \;\&\; (t_k \in F \rightarrow \Psi(t_k) = \Psi(r)) \;\&$$
$$\forall a \in \Sigma \; ((\neg!\mu(t_k, a) \;\&\; \neg!\mu(r, a)) \vee (!\mu(t_k, a) \;\&\; !\mu(r, a) \;\&$$
$$\mu(t_k, a) = \mu(r, a) \;\&\; \lambda(t_k, a) = \lambda(r, a))).$$

Theorem 3. *Let the subsequential transducer* $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ *be minimal except for* $\omega' = w_1 w_2 \dots w_m$. *Let* $\psi \in L(\mathcal{T})$ *be the last word in the lexicographical order of the input language of the transducer. Let* ω *be a word which is greater in lexicographical order than* ψ. *Let the* $\tau \in \Delta^*$ *be the output for* ω. *Let* ω' *be the longest common prefix of* ψ *and* ω. *In that case we can denote* $\omega = w_1 w_2 \dots w_m w_{m+1} \dots w_k; k > m$. *Let us use* W_n *to denote the word* $W_n = w_1 w_2 \dots w_n$ *; n* $= 1, 2, \dots, k$ *and* $W_0 = \varepsilon$. *Let us use* Λ_n *to denote the word* $\Lambda_n = \lambda^*(t_0, W_n) \wedge \tau$. *Let us define the subsequential transducer* $\mathcal{T}' = \langle \Sigma, S', s, F', \mu', \lambda', \Psi' \rangle$ *as follows:*

$t_{m+1}, t_{m+2}, \dots, t_k$ *are new states such that* $S \cap \{t_{m+1}, t_{m+2}, \dots, t_k\} = \emptyset$

$$S' = S \cup \{t_{m+1}, t_{m+2}, \dots, t_k\}$$

$$F' = F \cup \{t_k\}$$

$$\mu'(r, a) = \begin{cases} t_{i+1} & \text{, in case } r = t_i, m \le i \le k-1, a = w_{i+1} \\ \mu(r, a) & \text{, in case } r \in S \text{ and } \mu(r, a) \text{ is defined and} \\ & r \ne t_m \vee a \ne w_{m+1} \\ \text{is not defined} & \text{otherwise} \end{cases}$$

$$\lambda'(r, a) = \begin{cases} \lambda(r, a) & \text{, in case } r = S \setminus \{t_0, t_1, \dots, t_k\} \\ & \vee (r = t_0 \;\&\; a \ne w_1) \\ [\Lambda_{n-1}]^{-1} \Lambda_n & \text{, in case } r = t_{n-1}, a = w_n, n = 1, 2, \dots, m \\ [\Lambda_n]^{-1} \lambda^*(t_0, W_n a) & \text{, in case } r = t_n, a \ne w_{n+1}, n = 1, 2, \dots, m \\ [\Lambda_m]^{-1} \tau & \text{, in case } r = t_m, a = w_{m+1} \\ \varepsilon & \text{, in case } r = t_n, a = w_{n+1}, n = m+1, \dots, k-1 \\ \text{is not defined} & \text{otherwise} \end{cases}$$

$$\Psi'(r) = \begin{cases} \Psi(r) & \text{, in case } r \notin \{t_1, t_2, \dots, t_k\} \;\&\; r \in F \\ \{\varepsilon\} & \text{, in case } r = t_k \\ [\Lambda_n]^{-1} \lambda^*(t_0, W_n) \cdot \Psi(r) & \text{, in case } r = t_n \in F, n = 1, 2, \dots, m \\ \text{is not defined} & \text{otherwise} \end{cases}$$

Then the subsequential transducer $\mathcal{T}'$ *is minimal except for* ω, *and the following holds:* $L(\mathcal{T}') = L(\mathcal{T}) \cup \{\omega\}$, $O_{\mathcal{T}'}|_{L(\mathcal{T})} = O_{\mathcal{T}}$ *and* $O_{\mathcal{T}'}(\omega) = \{\tau\}$.

Theorem 4. *Let the subsequential transducer* $\mathcal{T} = \langle \Sigma, \Delta, S, s, F, \mu, \lambda, \Psi \rangle$ *be minimal except for* $\omega = w_1 w_2 \dots w_k$ *and* $\omega \in L(\mathcal{T})$ *be the last word in the lexicographical order of the input language of the transducer. Let* $\tau \in \Delta^*$ *be a new output for* ω*, such that* $\tau \notin O_{\mathcal{T}}(\omega)$*. Let us use* W_n *to denote the word* $W_n = w_1 w_2 \dots w_n$ *;* $n = 1, 2, \dots, k$ *and* $W_0 = \varepsilon$*. Let us use* Λ_n *to denote the word* $\Lambda_n = \lambda^*(t_0, W_n) \wedge \tau$*. Let us define the subsequential transducer* $\mathcal{T}' = \langle \Sigma, S, s, F, \mu, \lambda', \Psi' \rangle$ *as follows:*

$$\lambda'(r,a) = \begin{cases} \lambda(r,a) & \text{, in case } r = S \setminus \{t_0, \dots, t_k\} \vee (r = t_0 \;\&\; a \neq w_1) \\ [\Lambda_{n-1}]^{-1}\Lambda_n & \text{, in case } r = t_{n-1}, a = w_n, n = 1, 2, \dots, k \\ [\Lambda_n]^{-1}\lambda^*(t_0, W_n a) & \text{, in case } r = t_n, a \neq w_{n+1}, n = 1, 2, \dots, k-1 \\ \text{is not defined} & \text{otherwise} \end{cases}$$

$$\Psi'(r) = \begin{cases} \Psi(r) & \text{, in case } r \notin \{t_1, t_2, \dots, t_k\} \;\&\; r \in F \\ [\Lambda_n]^{-1}\lambda^*(t_0, W_n) \cdot \Psi(t_n) & \text{, in case } r = t_n \in F, n = 1, 2, \dots, k-1 \\ [\Lambda_k]^{-1}\lambda^*(t_0, W_k) \cdot \Psi(t_k) \cup & \\ \cup \{[\Lambda_k]^{-1}\tau\} & \text{, in case } r = t_k \\ \text{is not defined} & \text{otherwise} \end{cases}$$

Then the subsequential transducer $\mathcal{T}'$ *is minimal except for* ω*, and the following holds:* $L(\mathcal{T}') = L(\mathcal{T})$*,* $O_{\mathcal{T}'}|_{L(\mathcal{T}) \setminus \{\omega\}} = O_{\mathcal{T}}|_{L(\mathcal{T}) \setminus \{\omega\}}$ *and* $O_{\mathcal{T}'}(\omega) = O_{\mathcal{T}}(\omega) \cup \{\tau\}$*.*

We can use the proving schema introduced in [4] to prove the lemmae and theorems for minimal except for a word subsequential transducer. The only difference is that we have to check that the resulting transducers are canonical.

We can use the following equations for an efficient computation of the functions λ' and ψ' for the last two theorems.

$$\langle c_1 = \lambda(t_0, w_1) \wedge \tau \;\;,\;\; l_1 = c_1^{-1}\lambda(t_0, w_1) \;\;,\;\; \tau_1 = c_1^{-1}\tau \rangle$$

$$\langle c_2 = (l_1\lambda(t_1, w_2)) \wedge \tau_1 \;\;,\;\; l_2 = c_2^{-1}(l_1\lambda(t_1, w_2)) \;\;,\;\; \tau_2 = c_2^{-1}\tau_1 \rangle$$

$$\langle c_3 = (l_2\lambda(t_2, w_3)) \wedge \tau_2 \;\;,\;\; l_3 = c_3^{-1}(l_2\lambda(t_2, w_3)) \;\;,\;\; \tau_3 = c_3^{-1}\tau_2 \rangle$$

$$\vdots$$

$$\langle c_m = (l_{m-1}\lambda(t_{m-1}, w_m)) \wedge \tau_{m-1}, l_m = c_m^{-1}(l_{m-1}\lambda(t_{m-1}, w_m)), \tau_m = c_m^{-1}\tau_{m-1} \rangle$$

We can calulate c_n, l_n, τ_n iteratively for $n = 1, 2, \dots, m$.

We can proove by induction that:

$$c_n = [\lambda^*(t_0, W_{n-1}) \wedge \tau]^{-1}(\lambda^*(t_0, W_n) \wedge \tau)$$

$$l_n = [\lambda^*(t_0, W_n) \wedge \tau]^{-1}\lambda^*(t_0, W_n)$$

$$\tau_n = [\lambda^*(t_0, W_n) \wedge \tau]^{-1}\tau$$

for $n = 1, 2, \dots, m$. Hence we have that:

$$\lambda'(t_{n-1}, w_n) = c_n$$

$$\lambda'(t_n, a) = l_n \lambda(t_n, a)$$

$$\Psi'(t_n) = l_n \cdot \Psi(t_n)$$

for $a \neq w_{n+1}, n = 1, 2, \ldots, m$, and

$$\lambda'(t_m, w_{m+1}) = \tau_m \text{ for Theorem 3, or}$$

$$\Psi'(t_k) = l_k \cdot \Psi(t_k) \text{ for Theorem 4.}$$

Now we can proceed with the description of our method for direct building of minimal subsequential transducer for a given sorted list of words.

Let a non-empty finite list of words L in lexicographical order be given. Let for every word in L the corresponding output is given. Let $\omega^{(i)}$ denotes the i-th word of the list and $\tau^{(i)}$ denotes the output of the i-th word. We start with the minimal canonical subsequential transducer which recognizes only the first word of the list and outputs the output for the first word. This transducer can be built trivially and is also minimal except for $\omega^{(1)}$. Using it as a basis we carry out an induction on the words of the list. Let us assume that the transducer $\mathcal{T}^{(n)}$ with language $L^{(n)} = \{\omega^{(i)} \mid i = 1, 2, \ldots, n\}$ has been built and that $\mathcal{A}^{(n)}$ is minimal except for $\omega^{(n)}$ and $O_{\mathcal{T}^{(n)}}(\omega^{(i)}) = \tau^{(i)}$ for $i = 1, 2, \ldots, n$. We have to build the Transducer $\mathcal{T}^{(n+1)}$ with language $L^{(n+1)} = \{\omega^{(i)} \mid i = 1, 2, \ldots, n+1\}$ which is minimal except for $\omega^{(n+1)}$ and $O_{\mathcal{T}^{(n+1)}}(\omega^{(i)}) = \tau^{(i)}$ for $i = 1, 2, \ldots, n+1$.

Let ω' be the longest common prefix of the words $\omega^{(n)}$ and $\omega^{(n+1)}$. Using several times Lemma 1 and Lemma 2 (corresponding to the actual case) we build the transducer $\mathcal{T}'$ which is equivalent to $\mathcal{T}^{(n)}$ and is minimal except for ω'. Now we can use Theorem 3 (or Theorem 4 if $\omega^{(n)} = \omega^{(n+1)}$) to build the transducer $\mathcal{T}^{(n+1)}$ with language $L^{(n+1)} = L^{(n)} \cup \{\omega^{(n+1)}\} = \{\omega^{(i)} \mid i = 1, 2, \ldots, n+1\}$ which is minimal except for $\omega^{(n+1)}$ and $O_{\mathcal{T}^{(n+1)}}(\omega^{(i)}) = \tau^{(i)}$ for $i = 1, 2, \ldots, n+1$.

In this way by induction we build the minimal except for the last word of the list transducer with language the list L and the given output. At the end using again Lemma 1 and Lemma 2 we build the transducer equivalent to the former one which is minimal except for the empty word. From Proposition 1 we have that it is the minimal subsequential transducer for the list L and corresponding output.

To distinguish efficiently between Lemma 1 and Lemma 2 we can use the condition given in Lemma 3. □

Example 2. Let us consider the following example. On Figure 1 the transducer minimal except jul with input language $\{apr, aug, dec, feb, jan, jul\}$ and output function $O(apr) = \{30\}$; $O(aug) = \{31\}$; $O(dec) = \{31\}$; $O(feb) = \{28, 29\}$; $O(jan) = \{31\}$; $O(jul) = \{31\}$ is given. After the application of Lemma 2 and Theorem 3 we will construct the transducer minimal except for jun where $O(jun) = \{30\}$. This transducer is given on Figure 2. In this way we are adding the next word with the corresponding output to the transducer.

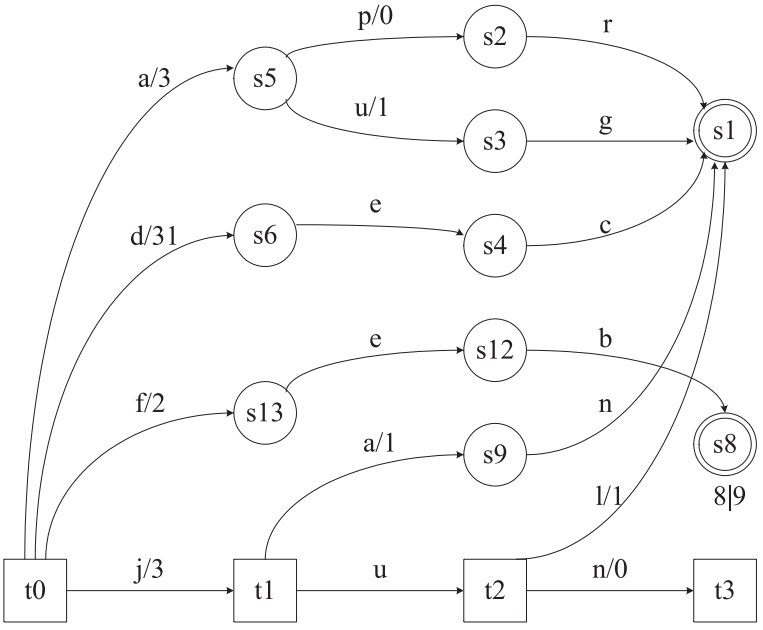


Fig. 2. Subsequential transducer minimal except for *jun*.

3 Algorithm for Building of Minimal Subsequential Transducer for a Given Sorted List

Here we give the pseudo-code in a Pascal-like language (like the language used in [1]). We will presume that there are given implementations for Abstract Data Types (ADT) representing transducer state and dictionary of transducer states. Later we presume that NULL is the null constant for arbitrary abstract data type.

On Transducer state we will need the following types and operations:

1. STATE is pointer to a structure representing a transducer state;
2. FIRST_CHAR, LAST_CHAR : are the first and the last char in the input alphabet;
3. **function** NEW_STATE : STATE returns a new state;
4. **function** FINAL(STATE) : boolean returns true if the state is final and false otherwise;
5. **procedure** SET_FINAL(STATE, boolean) sets the finality of the state to the boolean parameter;
6. **function** TRANSITION(STATE, char) : STATE returns the state to which the transducer transits from the parameter state with the parameter char;
7. **procedure** SET_TRANSITION(STATE, char, STATE) that sets the transition from first parameter state by the parameter char to the second parameter state;

8. **function** STATE_OUTPUT(STATE) : **set of** string returns the output set of strings on final states;
9. **procedure** SET_STATE_OUTPUT(STATE, **set of** string) sets the output set of strings on final states;
10. **function** OUTPUT(STATE, char) : string returns the output string for the transition from the parameter state by the parameter char;
11. **procedure** SET_OUTPUT(STATE, char, string) sets the output string for the transition from the parameter state by the parameter char;
12. **procedure** PRINT_TRANSDUCER(file, STATE) prints the transducer starting from the parameter state to file.

Having defined the above operations we make use of the following three functions and procedures:

1. **function** COPY_STATE(STATE) : STATE copies a state to a new one;
2. **procedure** CLEAR_STATE(STATE) clears all transitions of the state and sets it to non final;
3. **function** COMPARE_STATES(STATE, STATE) : integer compares two states

The ADT on Dictionary of transducer states uses the COMPARE_STATES function above to compare states. For the dictionary we need the following operations:

1. **function** NEW_DICTIONARY : DICTIONARY returns a new empty dictionary;
2. **function** MEMBER(DICTIONARY,STATE) : STATE returns state in the dictionary equivalent to the parameter state or NULL if not present;
3. **procedure** INSERT(DICTIONARY,STATE) inserts state to dictionary.

Implementation for the above ADTs could be found in e.g. [1]. Now we are ready to present the pseudo-code of our algorithm.

Algorithm 5. *For direct building of minimal subsequential transducer presenting the input list of words given in lexicographical order with their corresponding outputs.*

```
1   program Create_Minimal_Transducer_for_Given_List ( input, output);
2   var
3       MinimalTransducerStatesDictionary : DICTIONARY;
4       TempStates : array [0..MAX_WORD_SIZE] of STATE;
5       InitialState : STATE;
6       PreviousWord, CurrentWord, CurrentOutput,
              WordSuffix, CommonPrefix : string;
7       tempString : string;
8       tempSet : set of string;
9       i, j, PrefixLengthPlus1 : integer;
10        c : char;
```

```
11      function FindMinimized ( s : STATE) : STATE;
12      {returns an equivalent state from the dictionary. If not present –
          inserts a copy of the parameter to the dictionary and returns it.}
13      var r : STATE:
14      begin
15          r := MEMBER(MinimalTransducerStatesDictionary,s);
16          if r = NULL then begin
17              r := COPY_STATE(s);
18              INSERT(r);
19          end;
20          return(r);
21      end; {FindMinimized}
22  begin
23      MinimalTransducerStatesDictionary := NEW_DICTIONARY;
24      for i := 0 to MAX_WORD_SIZE do
25          TempState[i] := NEW_STATE;
26      PreviousWord := '';
27      CLEAR_STATE(TempState[0]);
28      while not eof(input) do begin
29      {Loop for the words in the input list}
30          readln(input,CurrentWord,CurrentOutput);
31          { the following loop calculates the length of the longest common
              prefix of CurrentWord and PreviousWord }
32          i := 1;
33          while (i<length(CurrentWord)) and (i<length(PreviousWord))
                and (PreviousWord[i] = CurrentWord[i]) do
34              i := i+1;
35          PrefixLengthPlus1 := i;
36          {we minimize the states from the suffix of the previous word }
37          for i := length(PreviousWord) downto PrefixLengthPlus1 do
38              SET_TRANSITION(TempStates[i-1],PreviousWord[i],
                        FindMinimized(TempStates[i]));
39          {This loop initializes the tail states for the current word}
40          for i := PrefixLengthPlus1 to length(CurrentWord) do begin
41              CLEAR_STATE(TempStates[i]);
42              SET_TRANSITION(TempStates[i-1],CurrentWord[i],
                        TempStates[i]);
43          end;
44          if CurrentWords <> PreviousWord then begin
45              SET_FINAL(TempStates[length(CurrentWord)], true);
46              SET_OUTPUT(TempStates[length(CurrentWord)], {''});
47          end;
48          for j := 1 to PrefixLengthPlus1-1 do begin
49              CommonPrefix := OUTPUT(TempStates[j-1], CurrentWord[j])
                        ∧ CurrentOutput;
```

```
50              WordSuffix := CommonPrefix^-1 OUTPUT(TempStates[j-1],
                      CurrentWord[j]);
51              SET_OUTPUT(TempStates[j-1], CurrentWord[j],
                      CommonPrefix);
52              for c := FIRST_CHAR to LAST_CHAR do begin
53                  if TRANSITION(TempStates[j],c) <> NULL then
54                      SET_OUTPUT(TempStates[j],c,concat(WordSuffix,
                              OUTPUT(TempStates[j],c)));
55              end;
56              if FINAL(TempStates[j]) then begin
57                  tempSet := ∅;
58                  for tempString in STATE_OUTPUT(TempStates[j]) do
59                      tempSet := tempSet ∪ concat(WordSuffix,tempString);
60                  SET_STATE_OUTPUT(TempStates[j], tempSet);
61              end;
62              CurrentOutput := CommonPrefix^-1 CurrentOutput;
63          end;
64          if CurrentWord = PreviousWord then
65              SET_STATE_OUTPUT(TempStates[length(CurrentWord)],
                    STATE_OUTPUT(TempStates[length(CurrentWord)])
                    ∪ CurrentOutput);
66          else SET_OUTPUT(TempStates[PrefixLengthPlus1-1],
                        CurrentWord[PrefixLengthPlus1],CurrentOutput);
67          PreviousWord := CurrentWord;
68      end; {while}
69      { here we are minimizing the states of the last word }
70      for i := length(CurrentWord) downto 1 do
71          SET_TRANSITION(TempStates[i-1],PreviousWord[i],
                    FindMinimized(TempStates[i]));
72      InitialState := FindMinimized(TempStates[0]);
73      PRINT_TRANSDUCER(output,InitialState);
74  end.
```

4 Implementation Results and Comparisons

Based on the main algorithm for direct building of minimal automata we have created implementation for direct construction of minimal automaton with labeled final states and minimal subsequential transducer. The results are summarized in the table bellow. We used a Bulgarian grammatical dictionary of simple words with about 900000 entries for the experiments. An implementation of the algorithm given in [3] has been used for the construction of the pseudo-minimal subsequential transducer.

In [6] Mehryar Mohri reports that the construction with his method of the p-subsequential transducer for a 672000 entries French dictionary takes 20' on a

Table 1. Comparison between different automata for the representation of a large-scale grammatical dictionary for Bulgarian.

Number of lines	895453		
Initial size	27.5 MB		
	Minimal Transducer	Pseudo-minimal Transducer	Minimal automaton with labeled final states
States	43413	531397	47854
Transitions	106809	992412	110791
Codes	16378	16378	6016
p	5	–	–
Size of codes	209K	–	126K
Size of automaton	1.3M	–	800K
Construction time	2'35"	–	25"
Memory used	5M	108M	2.5M

HP/9000 755 computer. All our experiments have been performed on a 500MHz Pentium III personal computer with 128MB RAM.

References

1. A. Aho, J. Hopcroft, J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, Massachutes, 1983.
2. J. Daciuk, S. Mihov, B. Watson, R. Watson, Incremental Construction of Minimal Acyclic Finite State Automata, *Computational Linguistics*, Vol. 26(1), 2000.
3. D. Maurel, Pseudo-minimal transducer, *Theoretical Computer Science*, Vol. 231(1), 129-139, 2000.
4. S. Mihov, Direct Building of Minimal Automaton for Given List, *Annuaire de l'Université de Sofia "St. Kl. Ohridski"*, Faculté de Mathématique et Informatique, volume 91, livre 1, 1998.
5. M. Mohri, Minimization of Sequential Transducers, *Lecture Notes in Computer Science*, Springer, Berlin, 1994.
6. M. Mohri, On Some Applications of Finite-State Automata Theory to Natural Language Processing, *Natural Language Engineering*, Vol. 2(1), 1-20, 1996.
7. M. Mohri, Finite-State Transducers in Language and Speech Processing, *Computational Linguistics*, Vol. 23(2), 269-311, 1997.
8. D. Revuz, *Dictionaires et lexiques – Méthodes et algorithmes*, Doctoral dissertation in Computer Science, University Paris VII, Paris, 1991.

Generic ϵ-Removal Algorithm for Weighted Automata

Mehryar Mohri

AT&T Labs – Research
180 Park Avenue, Rm E147
Florham Park, NJ 07932, USA mohri@research.att.com

Abstract. We present a new generic ϵ-removal algorithm for weighted automata and transducers defined over a semiring. The algorithm can be used with any semiring covered by our framework and works with any queue discipline adopted. It can be used in particular in the case of unweighted automata and transducers and weighted automata and transducers defined over the tropical semiring. It is based on a general shortest-distance algorithm that we briefly describe. We give a full description of the algorithm including its pseudocode and its running time complexity, discuss the more efficient case of acyclic automata, an on-the-fly implementation of the algorithm and an approximation algorithm in the case of the semirings not covered by our framework. We also illustrate the use of the algorithm with several semirings.

1 Introduction

Weighted automata are efficient and convenient devices used in many applications such as text, speech and image processing [6,2]. The automata obtained in such applications are often the result of various complex operations, some of them introducing the empty string ϵ. For example, using the classical method of Thompson, one can construct in linear time and space a non-deterministic automaton with ϵ-transitions representing a regular expression [9].

For the most efficient use of an automaton, it is preferable to *remove* the ϵ's of automata since in general they induce a delay in their use. An algorithm that constructs an automaton B with no ϵ's equivalent to an input automaton A with ϵ's is called *ϵ-removal*.

Textbooks do not present ϵ-removal of (unweighted) automata as an independent algorithm deserving a specific study. Instead, the algorithm is often mixed with other optimization algorithms such as determinization [1]. This usually makes the presentation of determinization more complex and the underlying ϵ-removal process obscure. Since ϵ-removal is not presented as an independent algorithm, it is usually not analyzed and its running time complexity not clearly determined.

We present a new generic ϵ-removal algorithm for weighted automata and transducers defined over a semiring. The algorithm can be used with any semiring covered by our framework and works with any queue discipline adopted. It can

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 230–242, 2001.

be used in particular in the case of unweighted automata and transducers and weighted automata and transducers defined over the tropical semiring. It is based on a general shortest-distance algorithm that we briefly describe. We give a full description of the algorithm including its pseudocode and its running time complexity, discuss the more efficient case of acyclic automata, an on-the-fly implementation of the algorithm and an approximation algorithm in the case of the semirings not covered by our framework. We also illustrate the use of the algorithm with several semirings.

2 Preliminaries

Weighted automata are automata in which the transitions are labeled with weights in addition to the usual alphabet symbols. For various operations to be well-defined, the weight set needs to have the algebraic structure of a semiring [5].

Definition 1. *A system* $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ *is a* right semiring *if:*

1. $(\mathbb{K}, \oplus, \overline{0})$ *is a commutative monoid with* $\overline{0}$ *as the identity element for* $\oplus$,
2. $(\mathbb{K}, \otimes, \overline{1})$ *is a monoid with* $\overline{1}$ *as the identity element for* $\otimes$,
3. $\otimes$ *right distributes over* $\oplus$: $\forall a, b, c \in \mathbb{K}, (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$,
4. $\overline{0}$ *is an annihilator for* $\otimes$: $\forall a \in \mathbb{K}, a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$.

Left semirings are defined in a similar way by replacing right distributivity with left distributivity. $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is a *semiring* if both left and right distributivity hold. Thus, more informally, a semiring is a ring that may lack negation. As an example, $(\mathbb{N}, +, \cdot, 0, 1)$ is a semiring defined on the set of nonnegative integers $\mathbb{N}$.

A semiring $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is said to be *idempotent* if for any $a \in \mathbb{K}$, $a \oplus a = a$. The boolean semiring $\mathcal{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$ and the tropical semiring $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ are idempotent, but $(\mathbb{N}, +, \cdot, 0, 1)$ is not.

Definition 2. *A* weighted automaton $A = (\Sigma, Q, I, F, E, \lambda, \rho)$ *over the semiring* $\mathbb{K}$ *is a 7-tuple where* Σ *is the finite alphabet of the automaton,* Q *is a finite set of states,* $I \subseteq Q$ *the set of initial states,* $F \subseteq Q$ *the set of final states,* $E \subseteq Q \times \Sigma \times \mathbb{K} \times Q$ *a finite set of transitions,* $\lambda : I \to \mathbb{K}$ *the initial weight function mapping* I *to* $\mathbb{K}$*, and* $\rho : F \to \mathbb{K}$ *the final weight function mapping* F *to* $\mathbb{K}$.

Given a transition $e \in E$, we denote by $i[e]$ its input 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, and by $E^R[q]$ the set of transitions entering 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 $p[\pi] = p[e_1]$. We denote by $P(q, q')$ the set of paths from q to q'. P can be extended to subsets $R \subseteq Q$ $R' \subseteq Q$, by:

$$P(R, R') = \bigcup_{q \in R,\, q' \in R'} P(q, q')$$

The labeling function i 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 $\otimes$-product of the weights of its constituent transitions:

$$i[\pi] = i[e_1] \cdots i[e_k]$$
$$w[\pi] = w[e_1] \otimes \cdots \otimes w[e_k]$$

Given a string $x \in \Sigma^*$, we denote by $P(x)$ the set of paths from I to F labeled with x:

$$P(x) = \{\pi \in P(I, F) : i[\pi] = x\}$$

The output weight associated by A to an input string $x \in \Sigma^*$ is:

$$A \cdot x = \bigoplus_{\pi \in P(x)} \lambda(p[\pi]) \otimes w[\pi] \otimes \rho(n[\pi])$$

If $P(x) = \emptyset$, $A \cdot x$ is defined to be $\overline{0}$. Note that weighted automata over the boolean semiring are equivalent to the classical unweighted finite automata.

These definitions can be easily generalized to cover the case of weighted automata with ϵ-transitions. An ϵ-removal algorithm computes for any input weighted automaton A with ϵ-transitions an equivalent weighted automaton B with no ϵ-transition, that is such that:

$$\forall x \in \Sigma^*, A \cdot x = B \cdot x$$

The following definitions will help us define the framework for our generic ϵ-removal algorithm [7].

Definition 3. *Let $k \geq 0$ be an integer. A commutative semiring $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is* k-closed *if:*

$$\forall a \in \mathbb{K},\ \bigoplus_{n=0}^{k+1} a^n = \bigoplus_{n=0}^{k} a^n$$

When $k = 0$, the previous expression can be rewritten as:

$$\forall a \in \mathbb{K}, \overline{1} \oplus a = \overline{1}$$

and $\mathbb{K}$ is then said to be *bounded*. Semirings such as the boolean semiring $\mathcal{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$ and the tropical semiring $\mathcal{T} = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ are bounded.

Definition 4. *Let $k \geq 0$ be an integer, $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ a commutative semiring, and A a weighted automaton over $\mathbb{K}$. $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is* right k-closed for A *if for any cycle π of A:*

$$\bigoplus_{n=0}^{k+1} w[\pi]^n = \bigoplus_{n=0}^{k} w[\pi]^n$$

By definition, if $\mathbb{K}$ is k-closed, then it is k-closed for any automaton over $\mathbb{K}$.

3 Algorithm

Let $A = (\Sigma, Q, I, F, E, \lambda, \rho)$ be a weighted automaton over the semiring $\mathbb{K}$ with ϵ-transitions. We denote by A_ϵ the automaton obtained from A by removing all transitions not labeled with ϵ. We present a general ϵ-removal algorithm for weighted automata based on a generic shortest-distance algorithm which works for any semiring $\mathbb{K}$ k-closed for A_ϵ [7]. In what follows, we assume that $\mathbb{K}$ has this property. This class of semirings includes in particular the boolean semiring $(\{0,1\}, \vee, \wedge, 0, 1)$, the tropical semiring $(\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$ and other semirings not necessarily idempotent. [1]

We present the algorithm in the case of weighted automata. The case of weighted transducers can be straightforwardly derived from the automata case by viewing a transducer as an automaton over the alphabet $\Sigma \cup \{\epsilon\} \times \Sigma \cup \{\epsilon\}$. An ϵ-transition of a transducer is then a transition labeled with (ϵ, ϵ). [2]

For p, q in Q, the ϵ-distance from p to q in the automaton A is denoted by $d[p, q]$ and defined as:

$$d[p, q] = \bigoplus_{\pi \in P(p,q),\, i[\pi] = \epsilon} w[\pi]$$

This distance is well-defined for any pair of states (p, q) of A when $\mathbb{K}$ is a semiring k-closed for A_ϵ [7]. By definition, for any $p \in Q$, $d[p, p] = \overline{1}$. $d[p, q]$ is the distance from p to q in A_ϵ.

3.1 Description and Proof

The algorithm works in two steps. The first step consists of computing for each state p of the input automaton A its ϵ-closure denoted by $C[p]$:

$$C[p] = \left\{(q, w) : q \in \epsilon[p],\, d[p, q] = w \in \mathbb{K} - \left\{\overline{0}\right\}\right\}$$

[1] The class of semirings with which our generic shortest-distance algorithm works can in fact be extended to that of right semirings *right k-closed for A_ϵ* [7], but for reasons of space we do not describe this more general framework here.

[2] We have also developed an algorithm for removing only input (or output) ϵ's of a weighted transducer without any modification to the alphabet of the transducer. That algorithm is also based on the generic shortest-distance algorithm presented in the next sections. It applies to all transducers that admit an equivalent input (or output) ϵ-free transducers. The complexity of the algorithm is exponential since in the worst case the size of the output transducer can be exponential in the input size.

where $\epsilon[p]$ represents the set of states reachable from p via a path labeled with ϵ. We describe this step in more detail later.

The second step consists of modifying the outgoing transitions of each state p by removing those labeled with ϵ and by adding to $E[p]$ non-ϵ-transitions leaving each state $q \in \epsilon[p]$ with their weights pre-$\otimes$-multiplied by $d[p, q]$. The following is the pseudocode of the second step of the algorithm.

ϵ-removal(A)
1 **for** each $p \in Q$
2 **do** $E[p] \leftarrow \{e \in E[p] : i[e] \neq \epsilon\}$
3 **for** each $(q, w) \in C[p]$
4 do $E[p] \leftarrow E[p] \cup \{(p, a, w \otimes w', r) : (q, a, w', r) \in E[q], a \neq \epsilon\}$
5 **if** $q \in F$
6 **then if** $p \notin F$
7 **then** $F \leftarrow F \cup \{p\}$
8 $\rho[p] \leftarrow \rho[p] \oplus (w \otimes \rho[q])$

State p is a final state if some state $q \in \epsilon[p]$ is final and the final weight $\rho[p]$ is then:

$$\rho[p] = \bigoplus_{q \in \epsilon[p] \cap F} (d[p, q] \otimes \rho[q])$$

Theorem 1. *Let $A = (\Sigma, Q, I, F, E, \lambda, \rho)$ be a weighted automaton over the semiring $\mathbb{K}$ right k-closed for A. Then the weighted automaton B result of the ϵ-removal algorithm just described is equivalent to A.*

Proof. We show that the function defined by the weighted automaton A is not modified by the application of one step of the loop of the pseudocode above. Let $A = (\Sigma, Q, I, F, E, \lambda, \rho)$ be the automaton just before the removal of the ϵ-transitions leaving state $p \in Q$ (lines 2-8).

Let $x \in \Sigma^*$, and let $Q(p)$ denote the set of successful paths labeled with x passing through p and either ending at p or following an ϵ-transition of p. By commutativity of $\oplus$, we have:

$$A \cdot x = \bigoplus_{\pi \in P(x) - Q(p)} \lambda(p[\pi]) \otimes w[\pi] \otimes \rho(n[\pi]) \oplus \bigoplus_{\pi \in Q(p)} \lambda(p[\pi]) \otimes w[\pi] \otimes \rho(n[\pi])$$

Denote by S_1 and S_2 the first and second term of that sum. The ϵ-removal at state p does not affect the paths not in $Q(p)$, thus we can limit our attention to the second term S_2. A path in $Q(p)$ can be factored in: $\pi = \pi_1 \pi_\epsilon \pi_2$, where π_ϵ is a portion of the path from p to $n[\pi_\epsilon] = q$ labeled with ϵ. The distributivity of $\otimes$ over $\oplus$ gives us:

$$S_2 = (\bigoplus_{n[\pi_1] = p} \lambda(p[\pi_1]) \otimes w[\pi_1])) \otimes S_{22}$$

with:

$$
\begin{aligned}
S_{22} &= \Big(\bigoplus_{p[\pi_\epsilon]=p,\, n[\pi_\epsilon]=q} w[\pi_\epsilon]\Big) \otimes \bigoplus_{p[\pi_2]=q} w[\pi_2] \otimes \rho(n[\pi_2]) \\
&= d[p,q] \otimes \bigoplus_{p[\pi_2]=q} w[\pi_2] \otimes \rho(n[\pi_2]) \\
&= \bigoplus_{p[\pi_2]=q} (d[p,q] \otimes w[\pi_2]) \otimes \rho(n[\pi_2])
\end{aligned}
$$

For paths π such that $\pi_2 = \epsilon$:

$$
S_{22} = \bigoplus_{p[\pi_2]=q,\, q \in \epsilon[p]} (d[p,q] \otimes \rho(q))
$$

which is exactly the final weight associated to p by ϵ-removal at p. Otherwise, by definition of π_ϵ, π_2 does not start with an ϵ-transition. The second term of the sum defining S_{22} can thus be rewritten as:

$$
S_{22} = \bigoplus_{e \in E[q],\, i[e] \neq \epsilon,\, \pi_2 = e\pi_2'} (d[p,q] \otimes w[e]) \otimes w[\pi_2'] \otimes \rho(n[\pi_2])
$$

The ϵ-removal step at state p follows exactly that decomposition. It adds non ϵ-transitions e leaving q with weights $(d[p,q] \otimes w[e])$ to the transitions leaving p. This ends the proof of the theorem. □

After removing ϵ's at each state p, some states may become inaccessible if they could only be reached by ϵ-transitions originally. Those states can be removed from the result in time linear in the size of the resulting machine using for example a depth-first search of the automaton.

Figures 1 (a)-(c) illustrate the use of the algorithm in the specific case of the tropical semiring. ϵ-transitions can be *removed* in at least two ways: in the way described above, or the reverse. The latter is equivalent to applying ϵ-removal to the reverse of the automaton and re-reversing the result. The two methods may lead to results of very different sizes as illustrated by the figures. This is due to two factors that are independent of the semiring:

- the number of states in the original automaton whose incoming transitions (or outgoing transitions in the reverse case) are all labeled with the empty string. As mentioned before, those states can be removed from the result. For example, state 3 of the automaton of figure 1 (a) can only be reached by ϵ-transitions and admits only outgoing transitions labeled with ϵ. Thus, that state does not appear in the result in both methods (figures 1 (b)-(c)). The incoming transitions of state 2 are all labeled with ϵ and thus it does not appear in the result of the ϵ-removal with the first method, but it does in the reverse method because the outgoing transitions of state 2 are not all labeled with ϵ;
- the total number of non ϵ-transitions of the states that can be reached from each state q in A_ϵ (the reverse of A_ϵ in the reverse case). This corresponds to the number of outgoing transitions of q in the result of ϵ-removal.

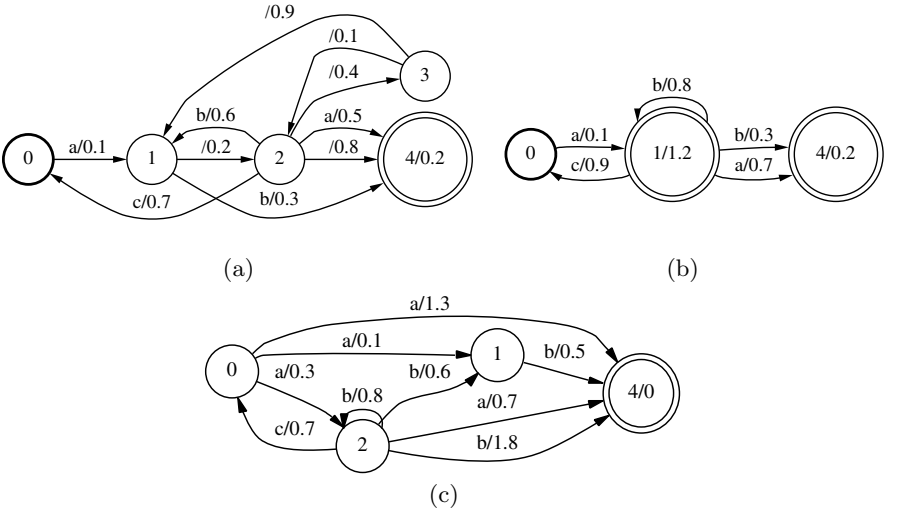


Fig. 1. ϵ-removal in the tropical semiring. (a) Weighted automaton A with ϵ-transitions. (b) Weighted automaton B equivalent to A result of the ϵ-removal algorithm. (c) Weighted automaton C equivalent to A obtained by application of ϵ-removal to the reverse of A.

In practice, one can use some heuristics to reduce the number of states and transitions of the resulting machine although this will not affect the worst case complexity of the algorithm. One can for instance remove some ϵ-transitions in the reverse way when that creates less transitions and others in the way corresponding to the first method when that helps reducing the resulting size.

Figures 2 (a)-(b) illustrate the algorithm in the case of another semiring, the semiring of real numbers. Our general algorithm applies in this case since A_ϵ is acyclic.

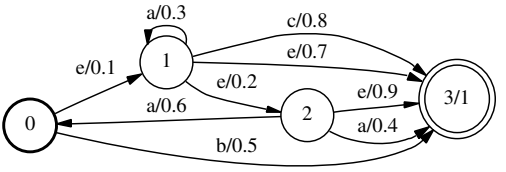
3.2 Computation of ϵ-Closures

As noticed before, the computation of ϵ-closures is equivalent to that of all-pairs shortest-distances over the semiring $\mathbb{K}$ in A_ϵ. There exists a generalization of the algorithm of Floyd-Warshall [3,11] for computing the all-pairs shortest-distances over a semiring $\mathbb{K}$ under some general conditions [7].[3] However, the running time complexity of that algorithm is cubic:

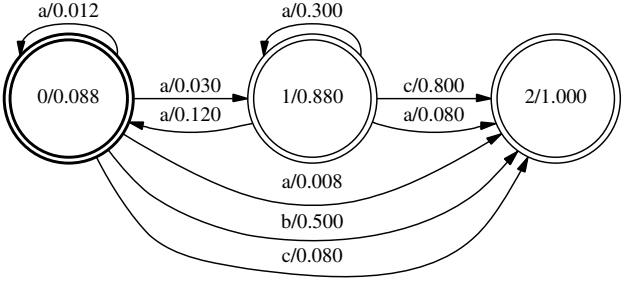
$$O(|Q|^3(T_\oplus + T_\otimes + T_*))$$

where $T_\oplus$, $T_\otimes$, and T_* denote the cost of $\oplus$, $\otimes$, and closure operations in the semiring considered. The algorithm can be improved by first decomposing A_ϵ

[3] That algorithm works in particular with the semiring $(\mathbb{R}, +, *, 0, 1)$ when the weight of each cycle of A_ϵ admits a well-defined closure.



(a)



(b)

Fig. 2. ϵ-removal in the real semiring $(\mathbb{R}, +, *, 0, 1)$. (a) Weighted automaton A. A_ϵ is k-closed for $(\mathbb{R}, +, *, 0, 1)$. (b) Weighted automaton B equivalent to A output of the ϵ-removal algorithm.

into its strongly connected components, and then computing all-pairs shortest distances in each components visited in reverse topological order. However, it is still impractical for large automata when A_ϵ has large cycles of several thousand states. The quadratic space complexity $O(|Q|^2)$ of the algorithm also makes it prohibitive for such large automata. Another problem with this generalization of the algorithm of Floyd-Warshall is that it does not exploit the sparseness of the input automaton.

There exists a generic single-source shortest-distance algorithm that works with any semiring covered by our framework [7]. The algorithm is a generalization of the classical shortest-paths algorithms to the case of the semirings of this framework.

This generalization is not trivial and does not require the semiring to be idempotent. In particular, a straightforward extension of the classical algorithms based on a relaxation technique would not produce the correct result in general [7]. The algorithm is also generic in the sense that it works with any queue discipline. The following is the pseudocode of the algorithm.

A queue S is used to maintain the set of states whose leaving transitions are to be relaxed. S is initialized to $\{s\}$ (line 4). For each state $q \in Q$, two attributes are maintained: $d[q] \in \mathbb{K}$ an estimate of the shortest distance from s to q, and $r[q] \in \mathbb{K}$ the total weight added to $d[q]$ since the last time q was extracted from

```
GENERIC-SINGLE-SOURCE-SHORTEST-DISTANCE (B, s)
 1  for each p ∈ Q
 2       do d[p] ← r[p] ← 0̄
 3  d[s] ← r[s] ← 1̄
 4  S ← {s}
 5  while S ≠ ∅
 6        do q ← head(S)
 7           DEQUEUE(S)
 8           r ← r[q]
 9           r[q] ← 0̄
10           for each e ∈ E[q]
11                do if d[n[e]] ≠ d[n[e]] ⊕ (r ⊗ w[e])
12                      then d[n[e]] ← d[n[e]] ⊕ (r ⊗ w[e])
13                           r[n[e]] ← r[n[e]] ⊕ (r ⊗ w[e])
14                           if n[e] ∉ S
15                              then ENQUEUE(S, n[e])
16  d[s] ← 1̄
```

Fig. 3. Generic single-source shortest-distance algorithm.

S. Lines $1-3$ initialize arrays d and r. After initialization, $d[q] = r[q] = \overline{0}$ for $q \in Q - \{s\}$, and $d[s] = r[s] = \overline{1}$.

Given a state $q \in Q$ and an transition $e \in E[q]$, a relaxation step on e is performed by lines 11-13 of the pseudocode, where r is the value of $r[q]$ just after the latest extraction of q from S if q has ever been extracted from S, its initialization value otherwise.

Each time through the **while** loop of lines 5-15, a state q is extracted from S (lines 6-7). The value of $r[q]$ just after extraction of q is stored in r, and then $r[q]$ is set to $\overline{0}$ (lines 8-9). Lines 11-13 relax each transition leaving q. If the tentative shortest distance $d[n[e]]$ is updated during the relaxation and if $n[e]$ is not already in S, the state $n[e]$ is inserted in S so that its leaving transitions be later relaxed (lines 14-15). $r[n[e]]$ is updated whenever $d[n[e]]$ is. $r[n[e]]$ stores the total weight $\oplus$-added to $d[n[e]]$ since $n[e]$ was last extracted from S or since the time after initialization if $n[e]$ has never been extracted from S. Finally, line 16 resets the value of $d[s]$ to $\overline{1}$.

In the general case, the complexity of the algorithm depends on the semiring considered and the queue discipline chosen for S:

$$O(|Q| + (T_{\oplus} + T_{\otimes} + C(A))|E| \max_{q \in Q} N(q) + (C(I) + C(E)) \sum_{q \in Q} N(q))$$

where $N(q)$ denotes the number of times state q is extracted from S, $C(E)$ the worst cost of removing a state q from the queue S, $C(I)$ that of inserting q in S, and $C(A)$ the cost of an assignment. [4]

In the case of the tropical semiring $(\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$, the algorithm coincides with classical single-source shortest-paths algorithms. In particular, it coincides with Bellman-Ford's algorithm when a FIFO queue discipline is used and with Dijkstra's algorithm when a shortest-first queue discipline is used. Using Fibonacci heaps [4], the complexity of Dijkstra's algorithm in the tropical semiring is:

$$O(|E| + |Q| \log |Q|)$$

The complexity of the algorithm is linear in the case of an acyclic automaton with a topological order queue discipline:

$$O(|Q| + (T_\oplus + T_\otimes)|E|)$$

Note that the topological order queue discipline can be generalized to the case of non-acyclic automata A_ϵ by decomposing A_ϵ into its strongly connected components. Any queue discipline can then be used to compute the all-pairs shortest distances within each strongly connected component [7].

The all-pairs shortest-distances of A_ϵ can be computed by running $|Q|$ times the generic single-source shortest-distance algorithm. Thus, when A_ϵ is acyclic, that is when A admits no ϵ-cycle, then the all-pairs shortest distances can be computed in quadratic time:

$$O(|Q|^2 + (T_\oplus + T_\otimes)|Q| \cdot |E|)$$

When A_ϵ is acyclic, the complexity of the computation of the all-pairs shortest distances can be substantially improved if the states of A_ϵ are visited in reverse topological order and if the shortest-distance algorithm is interleaved with the actual removal of ϵ's. Indeed, one can proceed in the following way. For each state p of A_ϵ visited in reverse topological order:

1. run a single-source shortest-distance algorithm with source p to compute the distance from p to each state q reachable from p by ϵ's;
2. remove the ϵ-transitions leaving q as described in the previous section.

The reverse topological order guarantees that the ϵ-paths leaving p are reduced to the ϵ-transitions leaving p. Thus, the cost of the shortest-distance algorithm run from p only depends on the number of ϵ-transitions leaving p and the total cost of the computation of the shortest-distances is linear:

$$O(|Q| + (T_\oplus + T_\otimes)|E|)$$

In the case of the tropical semiring and using Fibonacci heaps, the complexity of the first stage of the algorithm is:

$$O(|Q| \cdot |E| + |Q|^2 \log |Q|)$$

[4] This includes the potential cost of reorganization of the queue to perform this assignment.

In the worst case, in the second stage of the algorithm each state q belongs to the ϵ-closure of each state p, and the removal of ϵ's can create in the order of $|E|$ transitions at each state. Hence, the complexity of the second stage of the algorithm is:

$$O(|Q|^2 + |Q| \cdot |E|)$$

Thus, the total complexity of the algorithm in the case of an acyclic automaton A_ϵ is:

$$O(|Q|^2 + (T_\oplus + T_\otimes)|Q| \cdot |E|)$$

In the case of the tropical semiring and with a non acyclic automaton A_ϵ, the total complexity of the algorithm is:

$$O(|Q| \cdot |E| + |Q|^2 \log |Q|)$$

4 Remarks and Experiments

We have fully implemented the ϵ-removal algorithm presented here and incorporated it in recent versions of the FSM library [8]. For some automata of about a thousand states with large ϵ-cycles, our implementation is up to 600 times faster than the previous implementation based on a generalization of the Floyd-Warshall algorithm.

An important feature of our algorithm is that it admits a natural on-the-fly implementation. Indeed, the outgoing transitions of state q of the output automaton can be computed directly using the ϵ-closure of q. However, with an on-the-fly implementation, a topological order cannot be used for the queue S even if A_ϵ is acyclic since this is not known ahead of the time. Thus, we have implemented both an off-line and an on-the-fly version of the algorithm.

The algorithms presented in textbooks often combine the classical powerset construction, or determinization, and ϵ-removal [1]. Each state of the resulting automaton corresponds to the ϵ-closure of a subset of states constructed as in determinization.

If one wishes to apply determinization after ϵ-removal, then the integration of ϵ-removal within determinization may often be more efficient. This is because to compute the ϵ-closure of a subset created by determinization, one can run a single shortest-distance algorithm from the states of that subset rather than a distinct one for each state of the subset. Since some states can be reached by several elements of the subset, the first method provides more sharing. This is also corroborated by experiments carried out on various automata in the context of natural language processing applications [10].

This integration of determinization and ϵ-removal can be extended to the weighted case if one replaces the classical unweighted determinization by weighted determinization [6] and if one uses the weighted ϵ-closure presented in the previous sections. It is however limited to the cases where the weighted automaton resulting from ϵ-removal is determinizable since in the weighted case this is not always guaranteed.

Notice, however, that in the integrated algorithm presented in textbooks, the computation of the ϵ-closure is limited to the use of a FIFO queue discipline [1]. Thus, it is a special case of our general algorithm which may be less efficient both in terms of complexity and in practice than using a shortest-first queue discipline as in Dijkstra's algorithm, or a topological order in the case of acyclic automata. Our experiments confirm that in the case of acyclic automata, the ϵ-removal based on a topological order queue discipline can be many times faster than the one based on a shortest-first queue discipline, which itself is faster than the one based on a FIFO queue discipline.

The algorithm we presented can be straightforwardly modified to remove transitions with a label a different from ϵ. This can be done for example by replacing ϵ by a new label and a by ϵ, applying ϵ-removal and then restoring original ϵ's.

The shortest-distance algorithm presented in the previous sections admits an approximation version where the equality of line 11 in the pseudocode of figure 3 is replaced by an approximate equality modulo some predefined constant δ [7]. This can be used to remove ϵ-transitions of a weighted automaton A over a semiring $\mathbb{K}$ such as $(\mathbb{R}, +, *, 0, 1)$, even when $\mathbb{K}$ is not k-closed for A_ϵ. Although the transition weights are then only approximations of the correct results, this may be satisfactory for many practical purposes such as speech processing applications. Furthermore, one can arbitrarily improve the quality of the approximation by reducing δ. As mentioned before, one can also use a generalization of the Floyd-Warshall algorithm to compute the result in an exact way in the case of machines A_ϵ with relatively small strongly connected components and when the weight of each cycle of A_ϵ admits a well-defined closure.

5 Conclusion

A generic algorithm for ϵ-removal of weighted automata was given. The algorithm works with any semiring covered by our framework. In particular, it can work with semirings that are not necessarily idempotent. It works with any weighted automaton or transducer over the tropical semiring or the boolean semiring. It was shown to be more efficient than the existing algorithm both in space and time complexity. Experiments confirm this improvement of efficiency in practice. The algorithm admits a natural on-the-fly implementation which can be combined with other on-the-fly algorithms such as determinization or composition of weighted automata to optimize weighted automata.

Acknowledgments. I thank Michael Riley for several discussions about this work.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley: Reading, MA, 1986.
2. K. Culik II and J. Kari. Digital images and formal languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, pages 599–616. Springer, 1997.
3. R. W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 18, 1968.
4. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved optimalization problems. *Communications of the ACM*, 34, 1987.
5. W. Kuich and A. Salomaa. *Semirings, Automata, Languages*. Number 5 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin-New York, 1986.
6. M. Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23:2, 1997.
7. M. Mohri. General Algebraic Frameworks and Algorithms for Shortest-Distance Problems. Technical Memorandum 981210-10TM, AT&T Labs - Research, 62 pages, 1998.
8. M. Mohri, F. C. N. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231:17–32, January 2000.
9. K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11, 1968.
10. G. van Noord. Treatment of epsilon moves in subset construction. *Computational Linguistics*, 26:1, 2000.
11. S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

An $O(n^2)$ Algorithm for Constructing Minimal Cover Automata for Finite Languages*

Andrei Păun, Nicolae Sântean, and Sheng Yu

Department of Computer Science, University of Western Ontario
London, Ontario, Canada N6A 5B7
{apaun, santean, syu}@csd.uwo.ca

Abstract. Cover automata were introduced in [1] as an efficient representation of finite languages. In [1], an algorithm was given to transform a DFA that accepts a finite language to a minimal deterministic finite cover automaton (DFCA) with the time complexity $O(n^4)$, where n is the number of states of the given DFA. In this paper, we introduce a new efficient transformation algorithm with the time complexity $O(n^2)$, which is a significant improvement from the previous algorithm.

1 Introduction

Finite languages have many practical applications [6,2]. However, the finite languages used in applications are generally very large, which need thousands or even millions of states if represented by deterministic finite automata (DFA) or similar structures. In [1], deterministic finite cover automata (DFCA) were introduced as an alternative representation of finite languages. Experiments have shown that, in many cases, DFCA are much smaller in size than their corresponding minimal DFA [5].

Let L be a finite language and l the length of the longest word(s) in L. Intuitively, a DFCA A for L is a DFA that accepts all words in L and possibly additional words of length greater than l. So, a word w is in L if and only if it is accepted by A (as a DFA) and it has a length less than or equal to l. Note that checking the length of a word is usually not an extra burden in practice since the length of an input word is kept anyway in most applications.

In order to explain intuitively the notion of a DFCA, we give a very simple example in the following. Let $\Sigma = \{a, b, c\}$ be the alphabet and $L = \{abc, ababc, abababc\}$ a finite language over Σ. Clearly, the length of the longest word in L is 7, i.e., $l = 7$. The minimal DFA accepting L is shown in Figure 1, which has 8 states (9 if complete). A minimal DFCA is shown in Figure 2, which has only 4 states (5 if complete).

In [1], an algorithm was given for constructing a minimal DFCA from a given DFA that accepts a finite language. The time complexity of the algorithm

* This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada grants OGP0041630 and a graduate scholarship.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 243–251, 2001.

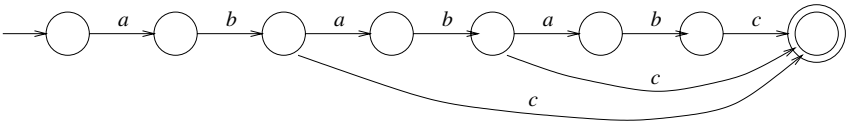


Fig. 1. The minimal DFA accepting L

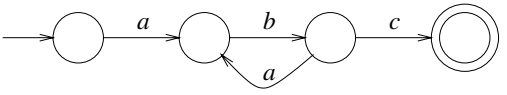


Fig. 2. A minimal DFCA for L with $l = 7$

is $O(n^4)$, where n is the number of states of the DFA. Note that the number of transitions of a DFA is linear to its number of states. In this paper, we give an $O(n^2)$ algorithm for the construction of a minimal DFCA from a given DFA. The new algorithm is not only a significant improvement from the previous algorithm [1] in time complexity, but also much easier to comprehend and to implement.

The two algorithms differ mainly at how to compute the similarity (or dissimilarity) relation between states. The new algorithm computes the pairs of states that are dissimilar and propagates the dissimilarity relations, rather than to compute directly the similarity relation as in the algorithm in [1]. A new algorithm is also given for merging similar states, which is simpler than the one given in [1]. We also prove several new theorems on the similarity relation which form the theoretical basis of the new algorithm.

In the next section, we give the basic definitions and notation, as well as the basic results, on cover languages and automata. In Section 3, we prove two theorems which are essential to the new algorithm. In Section 4, we describe our new algorithm and analyze its complexity. In the last section, we conclude the paper.

2 Preliminaries

First, we give the basic definitions and notation for cover languages, cover automata, and the similarity relation. Then we list some basic results, which are relevant to this paper, without giving any proofs. Detailed explanations and proofs can be found in [1] or [5].

Let S be a finite set and n a nonnegative integer. By $S^{\leq n}$ we denote $\cup_{i=0}^{n} S^i$.

Definition 1. *Let $L \subset \Sigma^*$ be a finite language over an alphabet Σ and l the length of the longest word(s) in L. A language L' over Σ is called a cover language of L if $L' \cap \Sigma^{\leq l} = L$.*

Definition 2. *A cover automaton for a finite language L is a finite automaton A such that the language accepted by A, i.e., $L(A)$, is a cover language of L. If A is a DFA, then A is called a deterministic finite cover automaton (DFCA) for L.*

We often use the term cover automaton casually to mean DFCA in this paper.

In the following, we give the basic definitions regarding the similarity relation. We first define the similarity relation on words respect to a finite language, and then the similarity relation on states of a DFA that accepts a finite language. The notion of similarity between words was first introduced in [4], and then studied in [3], [1], [5], etc. The concept of the similarity relation on words is the basis for the similarity relation on states of a DFA.

Definition 3. *Let L be a finite language over the alphabet Σ and l the length of the longest word(s) in L. Let $x, y \in \Sigma^*$. We define the following relation:*

(1) $x \sim_L y$ if for all $z \in \Sigma^*$ such that $\mid xz \mid \leq l$ and $\mid yz \mid \leq l, xz \in L$ iff $yz \in L$;
(2) $x \not\sim_L y$ if $x \sim_L y$ does not hold.

The relation $\sim_L$ is called the *similarity* relation with respect to L. We will use $x \sim y$ instead of $x \sim_L y$ when L is clearly understood from the context. Note that the relation $\sim_L$ is reflexive, symmetric, but NOT transitive.

Lemma 1. *Let $L \subseteq \Sigma^*$ be a finite language and $x, y, z \in \Sigma^*$, $|x| \leq |y| \leq |z|$. The following statements hold:*

1. *If $x \sim_L y$, $x \sim_L z$, then $y \sim_L z$.*
2. *If $x \sim_L y$, $y \sim_L z$, then $x \sim_L z$.*
3. *If $x \sim_L y$, $y \not\sim_L z$, then $x \not\sim_L z$.*

Definition 4. *Let $L \in \Sigma^*$ be a finite language.*

1. *A sequence of words $(x_1, \ldots, x_n)$ over Σ is called a dissimilar sequence of L if $x_i \not\sim_L x_j$ for each pair i, j, $1 \leq i, j \leq n$ and $i \neq j$.*
2. *A dissimilar sequence $(x_1, \ldots, x_n)$ of L is called a maximal dissimilar sequence of L if for any dissimilar sequence $(y_1, \ldots, y_m)$ of L, $m \leq n$.*

In the following, we define the similarity relation on the set of states of a DFA or a DFCA. Note that if a DFA A accepts a finite language L, then A is also a DFCA for L.

Definition 5. *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA (or a DFCA). We define, for each state $q \in Q$,*

$$level(q) = min\{|w| \mid \delta(s, w) = q\},$$

i.e., $level(q)$ is the length of the shortest path (in the directed graph associated with the automaton) from the initial state to q.

Definition 6. *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFCA for a finite language L with l being the longest word(s) in L. Let $p, q \in Q$ and $m = \max\{level(p), level(q)\}$. We say that $p \sim_A q$ if for every $w \in \Sigma^{\leq l-m}$, $\delta(p, w) \in F$ iff $\delta(q, w) \in F$.*

We use the notation $p \sim q$ instead of $p \sim_A q$ whenever L is clearly understood from the context.

We are now ready to state the theorem that is the basis for any algorithm for the minimization of DFCAs.

Theorem 1. *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFCA for a finite language L. Assume that $p \sim_L q$ for some $p, q \in Q$ such that $p \neq q$ and $level(p) \leq level(q)$. Then we can construct a DFCA $A' = (Q', \Sigma, \delta', s, F')$ for L such that $Q' = Q - \{q\}$, $F' = F - \{q\}$, and*

$$\delta'(t, a) = \begin{cases} \delta(t, a) & \text{if } \delta(t, a) \neq q, \\ p & \text{otherwise} \end{cases}$$

for each $t \in Q'$ and $a \in \Sigma$.

Definition 7. *A DFCA A for a finite language is a minimal DFCA if and only if no two different states of A are similar.*

Theorem 2. *For a finite language L, there is a unique number $N(L)$ such that any minimal DFCA for L has exactly $N(L)$ states.*

Please refer to [1] for the proofs of the above theorems.

Definition 8. *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA or a DFCA for a finite language L with l be the length of the longest word(s) in L. For $p \in Q$, denote by x_p a shortest word in Σ^* such that $\delta(s, x_p) = p$; x_p is called a "representative" of p.*

Note that for each $q \in Q$, $|x_q| = level(q)$.

Theorem 3. *$p \sim q$ if and only if $x_p \sim x_q$.*

One may refer to [1] for a proof.

3 The New Algorithm

Given a DFA that accepts a finite language, we can construct a minimal DFCA for the given language in two steps: (1) compute the similarity relation between the states of the DFA, and (2) merge similar states. Note that the similarity relation is not transitive. So, if $p \sim q$ and $q \sim r$, we cannot simply merge p, q, and r together in general. Step (1) is the most complex one. A naive algorithm for determine whether $p \sim q$ is to check whether $\delta(p, z), \delta(q, z) \in F$ or $\delta(p, z), \delta(q, z) \in Q - F$ for all words z such that $|z| \leq l - \max(level(p), level(q))$. This would need exponential time. In the algorithm given in [1], it needs $O(n^2)$

time to determine whether two states are similar. The time complexity of the entire step (1) of that algorithm is $O(n^4)$.

Here we use a different approach and the time complexity of our algorithm is $O(n^2)$. In this section, we will describe our new algorithm. However, before describing the algorithm, we have to give several new definitions and prove several new results.

3.1 New Definitions and Results

Again we assume that $A = (Q, \Sigma, \delta, s, F)$ is a DFA accepting a finite language L over the alphabet Σ and l is the length of the longest word(s) in L. We assume that A is a complete DFA and there is no useless state in Q except the sink state d, i.e., for each $q \in Q - \{d\}$, there exist $u, v \in \Sigma^*$ such that $\delta(s, u) = q$ and $\delta(q, v) \in F$.

Definition 9. *For $p, q \in Q$ and $p \neq q$, we define*

$$range(p, q) = l - \max\{level(p), level(q)\}.$$

Intuitively, $range(p, q)$ is the maximum length of a word w that satisfies both $|x_p w| \leq l$ and $|x_q w| \leq l$.

Definition 10. *Let $p, q \in Q$ and $z \in \Sigma^*$. We say that p and q fail on z if $\delta(p, z) \in F$ and $\delta(q, z) \in Q - F$ or vice versa, and $|z| \leq range(p, q)$.*

Theorem 4. *$p \not\sim q$ if and only if there exists $z \in \Sigma^*$ such that p and q fail on z.*

Definition 11. *If $p \not\sim q$, we define*

$$gap(p, q) = min\{|z| \mid p \text{ and } q \text{ fail on } z\}.$$

If $p \not\sim q$, then $gap(p, q)$, intuitively, is the length of the shortest word(s) that can show that p and q are dissimilar. It is clear that $gap(p, q) = gap(q, p)$ and $gap(p, q) < l$ for any $p, q \in Q$ such that $p \not\sim q$. For convenience, we define $gap(p, q) = l$ if $p \sim q$. The next theorem is clear.

Theorem 5.

(1) Let d be the sink state of A. If $level(d) > l$, then $d \sim q$ for each $q \in Q - \{d\}$. If $level(d) \leq l$, then $d \not\sim f$ and $gap(d, f) = 0$ for each $f \in F$.
(2) If $p \in F$ and $q \in Q - F - \{d\}$ or vice versa, then $p \not\sim q$ and $gap(p, q) = 0$.

Lemma 2. *Let $p, q \in Q$, $p \neq q$, and $r = \delta(p, a)$ and $t = \delta(q, a)$, for some $a \in \Sigma$. Then $range(p, q) \leq range(r, t) + 1$.*

Proof. It is clear that $level(r) \leq level(p) + 1$ and $level(t) \leq level(q) + 1$. So,

$$\max(level(r), level(t)) \leq \max(level(p), level(q)) + 1.$$

Then, by Definition 9, $range(p, q) \leq range(r, t) + 1$. □

Theorem 6. *Let p and q be two states such that either $p, q \in F$ or $p, q \in Q - F$. Then $p \not\sim q$ if and only if there exists $a \in \Sigma$ such that $\delta(p, a) = r$ and $\delta(q, a) = t$, $r \not\sim t$, and*

$$gap(r, t) + 1 \leq range(p, q).$$

Proof. Only if: We assume that $p \not\sim q$ and will show that there exists a pair (r, t) satisfying the conditions of the theorem. Choose $z \in \Sigma^*$ such that p and q fail on z and $|z| = gap(p, q)$. Note that $|z| > 0$ because of the given condition of p and q. By the definition of *gap* function, we know that $|z| \leq range(p, q)$. Without loss of generality, we assume that $\delta(p, z) \in F$ and $\delta(q, z) \in Q - F$. Let $z = az'$. Then $\delta(p, az') = \delta(r, z') \in F$ and $\delta(q, az') = \delta(t, z') \in Q - F$ for some $r, t \in Q$. By Lemma 2, we know that $range(r, t) \geq range(p, q) - 1$. Then $|z'| \leq range(r, t)$. By Definition 10, r and t fail on z' and $r \not\sim t$. Since $gap(r, t) \leq |z'|$, we have $gap(r, t) + 1 \leq range(p, q)$.

If: Assume that there exists $a \in \Sigma$ such that $\delta(p, a) = r$, $\delta(q, a) = t$, $r \not\sim t$, and $gap(r, t) + 1 \leq range(p, q)$. Then there is $z' \in \Sigma^*$ such that r and t fail on z' and $|z'| = gap(r, t)$. Let $z = az'$. Then $|z| = gap(r, t) + 1$ and thus $|z| \leq range(p, q)$. Therefore, p and q fail on z. In other words, $p \not\sim q$. □

The following theorem gives a formula which computes $gap(p, q)$ for two state p and q that are either both final states or both non-final states.

Theorem 7. *If $p \not\sim q$ such that $p, q \in F$ or $p, q \in Q - F$, then*

$$gap(p, q) = \min\{gap(r, t) + 1 \mid \delta(p, a) = r \text{ and } \delta(q, a) = t, \text{ for } a \in \Sigma, \\ r \not\sim t, \text{ and } gap(r, t) + 1 \leq range(p, q)\}.$$

Proof. We first prove that $gap(p, q) \leq gap(r, t) + 1$ for every pair $r, t \in Q$ such that $\delta(p, a) = r$, $\delta(q, a) = t$, $r \not\sim t$, and $gap(r, t) + 1 \leq range(p, q)$. Let (r, t) be an arbitrary pair that satisfy the above conditions. Since $r \not\sim t$, there exists z' such that r and t fail on z' and $|z'| = gap(r, t)$. It is also clear that $|az'| \leq range(p, q)$ since $gap(r, t) + 1 \leq range(p, q)$. Then p and q fail on $z = az'$. By definition, $gap(p, q) \leq |z|$. So, we have $gap(p, q) \leq gap(r, t) + 1$.

We now prove the other direction, i.e., there exist $r, t \in Q$ such that $\delta(p, a) = r$, $\delta(q, a) = t$, $r \not\sim t$, and $gap(r, t) + 1 \leq gap(p, q)$. Let $z \in \Sigma^*$ such that p and q fail on z and $|z| = gap(p, q)$. Clearly, $|z| > 0$ by the given conditions. Then $z = az'$ for some $a \in \Sigma$. Let $\delta(p, a) = r$ and $\delta(q, a) = t$. Then $range(p, q) \leq range(r, t) + 1$ by Lemma 2. Thus, $|z'| \leq range(r, t)$. Then clearly r and t fail on z'. By definition, $gap(r, t) \leq |z'|$. Then we have $gap(r, t) \leq |z| - 1 = gap(p, q) - 1$. □

3.2 The Algorithm

The algorithm consists of two main parts: the first is to determine the similarity relation between states; the second is to merge similar states.

In the first part of the algorithm, we determine the similarity relation by computing the *gap* function starting from the sink state and along the inverse direction of the transitions of the given DFA. Note that the construction of a minimal DFCA is different from the minimization of a acyclic DFA. In the latter case, if two states have different *heights* then they are not equivalent. (The height of a state is the length of the longest path starting from this state to a final state.) However, in the former, it is possible that two states are similar even if they have different heights. The equivalence relation is a refinement of the similarity relation with respect to a finite language.

In the following, we assume that the given DFA accepting a finite language is complete (a transition is defined for each state and each letter in the alphabet) and reduced (no useless states except one sink state). We also assume that the given DFA is ordered, i.e., the $n+1$ states (including the sink state) of the DFA are numbered by $0, 1, \ldots, n$ such that there is no transition from state j to state i if $0 \leq i < j \leq n$. This implies that 0 is the starting state, n is the sink state, and *n-1* is the last final state. All the above pre-conditions can be achieved in linear time in terms of the number of states of the given DFA. Note that the size of a DFA is linear to its number of states.

Algorithm for computing the *gap* function

Input: An ordered, reduced, and complete DFA $A = (Q, \Sigma, \delta, 0, F)$, with $n+1$ states, which accepts a finite language L, and the length l of the longest word in L

Output: $gap(i, j)$ for each pair $i, j \in Q$ and $i < j$

Algorithm:

1. For each $i \in Q$ compute $level(i)$ end for;
2. for $i = 0$ to $n - 1$ do $gap(i, n) = l$ end for;
   if $level(n) \leq l$ then
   for each $i \in F$ $gap(i, n) = 0$ end for
   end if;
3. for each pair $i, j \in Q - \{n\}$ such that $i < j$
   if $i \in F$ and $j \in Q - F$ or vice versa then
   $gap(i, j) = 0$;
   else
   $gap(i, j) = l$;
   end if;
   end for;
4. for $i = n - 2$ down to 0 do
   for $j = n$ down to $i + 1$ do
   for each $a \in \Sigma$ do
   let $i' = \delta(i, a)$ and $j' = \delta(j, a)$;
   if $i' \neq j'$ then
   $g =$ if $(i' < j')$ then $gap(i', j')$ else $gap(j', i')$;

```
                        if g + 1 ≤ range(i, j) then
                            gap(i,j) = min(gap(i,j), g+1);
                        end if;
                    end if;
                end for;
            end for;
        end for
```

Algorithm for merging similar states
Input: A ordered, reduced, and complete DFA $A = (Q, \Sigma, \delta, 0, F)$ which accepts a finite language L, and $gap(i, j)$ for each pair $i, j \in Q$ and $i < j$
Output: A minimal DFCA A' for L
Algorithm:

1. Let $P[0..n]$ be a Boolean array with each $P[i]$, $0 \leq i \leq n$, initialized to $false$;
2. for $i = 0$ to $n - 1$ do
        if $P[i] == false$ then
            for $j = i + 1$ to n do
                if $P[j] == false$ and $gap(i, j) = l$ then
                    merge j to i;
                    P[j] = $true$;
                end if;
            end for;
        end if;
    end for.

For convenience, we assume that the number of states is $n + 1$ in the above algorithm and there is at least one state in A. Thus $n = 0$ if there is only one state in A.

The step "merge j to i;" follows the steps described in Theorem 1.

The correctness of the algorithm can be easily established with Theorem 5, Theorem 6, and Theorem 7. So, we omit the formal proof here.

We now consider the time complexity of the algorithm. In the first part, each of Step 1 and Step 2 is $O(n)$. Clearly, Step 3 takes $O(n^2)$ iterations. Step 4 is the main part, which has two nested loops, each of which has $O(n)$ iterations. Each inner iteration is $O(|\Sigma|)$, where $|\Sigma|$ is a constant. Therefore, the first part of the algorithm, that computes the *gap* function, is $O(n^2)$. Clearly, the second part is also $O(n^2)$. So, the time complexity of the algorithm is $O(n^2)$.

4 Concluding Remarks

We have shown an $O(n^2)$ algorithm for constructing a minimal DFCA for a finite language given in the form of a DFA. This is a significant improvement from the $O(n^4)$ algorithm given in [1]. This new algorithm is also much easy to

comprehend and implement. The algorithm can be modified into a minimization algorithm for general DFCA.

In the future, we will conduct more experiments on DFCA with finite languages from real-world applications. It is important to know how much reduction on the size of the automata one can achieve by using DFCA instead of DFA. We believe that the reduction can be large for certain types of applications, but minor on others.

References

1. C. Campeanu, N. Santean, S. Yu, "Minimal Cover-Automata for Finite Languages", *Proceedings of the Third International Workshop on Implementing Automata* (WIA'98) 1998, 32-42.
2. J.-M. Champarnaud and D. Maurel, *Automata Implementation*, Third International Workshop on Implementing Automata, LNCS 1660, Springer, 1999.
3. C. Dwork and L. Stockmeyer, "A Time Complexity Gap for Two-Way Probabilistic Finite-State Automata", *SIAM Journal on Computing*, vol.19 (1990) 1011-1023.
4. J. Kaneps, R. Frievalds, "Running Time to Recognize Non-Regular Languages by 2-Way Probabilistic Automata", *in ICALP'91, LNCS*, Springer-Verlag, New-York/Berlin (1991) vol 510, 174-185.
5. N. Santean, *Towards a Minimal Representation for Finite Languages: Theory and Practice*, MSc Thesis, Department of Computer Science, The University of Western Ontario, 2000.
6. D. Wood and S. Yu, *Automata Implementation*, Second International Workshop on Implementing Automata, LNCS 1436, Springer, 1998.

Unary Language Concatenation and Its State Complexity*

Giovanni Pighizzini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39 – 20135 Milano, Italy
pighizzi@dsi.unimi.it

Abstract. In this paper we study the costs, in terms of states, of some basic operations on regular languages, in the unary case, namely in the case of languages defined over a one letter alphabet. In particular, we concentrate our attention on the concatenation. The costs, which are proved to be tight, are given by explicitly indicating the number of states in the noncyclic and in the cyclic parts of the resulting automata.

1 Introduction

Finite automata are one of the first computational models presented in the literature and, certainly, one of the most extensively investigated. However, some problems concerning these simple models are still open and the investigation of some aspects of the finite automata world is only at the beginning.

For instance, many complexity results for finite automata are given under the hypothesis that the input alphabet contains at least two symbols. As an example, consider the simulation of an n–state nondeterministic automaton by an equivalent 2^n–state deterministic one: it is optimal when the input alphabet contains at least two symbols [6], but, as shown in 1986 by Chrobak [2], its cost can be reduced in the *unary* case, namely for automata with a one letter input alphabet.

Very recently, the investigation of the unary case was reconsidered by pointing out many differences between the world of unary automata and the universe of all finite automata [7]. More generally, differences between the unary and the general case have been shown not only for finite automata and regular languages, but even for other classes of machines and languages (see, e.g., [3,5,9]).

In this paper, we study the *state complexity* of some operations on unary languages. We recall that state complexity is a type of descriptional complexity for regular languages, based on deterministic finite automata. More precisely, the state complexity of a regular language is the number of states of the minimum automaton accepting it. Furthermore, as pointed out by Shallit and Breitbart

* Partially supported by MURST, under the project "Modelli di calcolo innovativi: metodi sintattici e combinatori".

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 252–262, 2001.

[10], state complexity can be extended in order to estimate even the costs of the descriptions of nonregular languages.

In order to evaluate the state complexity of an operation as, for instance, the concatenation, we have to express the number of states of the minimum deterministic automaton accepting the concatenation of two languages L' and L'', as a function of the numbers of states of the minimal deterministic automata accepting L' and L''.

Tight evaluations of the state complexity of the concatenation and of the star of regular languages were obtained by Yu, Zhuang, and Salomaa [13]. For both operations, the state complexity of the resulting language can be exponential. On the other hand, as pointed out in [12], the state complexity of the union and of the intersection of two regular languages is the product of their state complexities. However, as shown in the same papers, if L' and L'' are *unary* languages accepted by two unary deterministic automata with m and n states, respectively, such that the numbers m and n are relatively prime, then the worst case state complexity of $L'L''$, $L' \cap L''$, and $L' \cup L''$ is mn, while the state complexity of L'^* is $(m-1)^2+1$. Recently, the same operations on unary regular languages were considered by C. Nicaud [8], obtaining estimations of their average state complexities.

Since the transition graph of a unary deterministic automaton A consists of an initial path of $\mu \geq 0$ states which is followed by a cycle of $\lambda \geq 1$ states (the pair (λ, μ) will be called *size* of A), to study the state complexity in the unary case it seems quite natural to taking into account not only the total number of states, but even how many of them are in the cyclic part and how many of them are in the initial path.

Recently, J. Shallit considered the union and the intersection of unary regular languages: he proved that if L' and L'' are accepted by two automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively, then both $L' \cup L''$ and $L' \cap L''$ are accepted by automata of size (λ, μ), where λ is the least common multiple of λ' and λ'', and μ is the maximum between μ' and μ''. Furthermore, this upper bound is tight [11]. In this paper, we consider the concatenation of L' and L''. In Section 3 we prove that the language $L'L''$ is accepted by an automaton of size (λ, μ), where λ is, as for the union and intersection, the least common multiple of λ' and λ'', while $\mu = \mu' + \mu'' + \lambda - 1$. We also consider some particular situations, e.g., all the states on the initial path of A' or of A'' are nonfinal, or λ' and λ'' are relatively prime. We prove that in these cases the size of the resulting automaton can be further reduced. For instance, when λ' and λ'' are relatively prime and both L', L'' are infinite, the length of the cycle in the minimum automaton accepting $L'L''$ reduces to one. With few exceptions, these estimations are shown to be tight, *for all possible sizes* of the given automata. In Table 1 and Table 2 at the end of Section 3, all these results concerning the state complexity of the concatenation are summarized.

We conclude the paper by presenting, in Section 4, some considerations concerning the star operation.

For brevity reasons, some of the proofs are omitted or just outlined in this version of the paper.

2 Preliminary Notions and Results

In this section, we recall basic notions, notations and facts used in the paper.

Given two integers $a, b > 0$, we denote by $\gcd(a, b)$ and by $\mathrm{lcm}(a, b)$, their *greatest common divisor* and their *least common multiple*, respectively. The following result will be crucial in order to evaluate the number of states of unary automata:

Lemma 1. *Given two integers $a, b > 0$, each number of the form $ax + by$, with $x, y \geq 0$, is a multiple of $\gcd(a, b)$. Furthermore, the largest multiple of $\gcd(a, b)$ that cannot be represented as $ax + by$, with $x, y \geq 0$, is $\mathrm{lcm}(a, b) - (a + b)$.*

Proof. It is well–known that each number $z = ax + by$ is a multiple of $g = \gcd(a, b)$. Let $a' = a/g$ and $b' = b/g$. Then $gcd(a', b') = 1$. The largest integer that cannot be represented as $a'x + b'y$, with $x, y \geq 0$, is $a'b' - (a' + b')$ (see, e.g., [13]). By just multiplying by g, we get that the largest multiple of g that cannot be written as $ax + by$, with $x, y \geq 0$, is $\mathrm{lcm}(a, b) - (a + b)$. □

Given an alphabet Σ, Σ^* denotes the set of strings on Σ. Given a language $L \subseteq \Sigma^*$, its complement, i.e., the set $\Sigma^* - L$, is denoted as L^c. A language L is said to be *unary* (or *tally*) whenever it can be built over a *single letter* alphabet. In this case, we let $L \subseteq 1^*$.

The computational models we will consider in this paper are *one–way deterministic finite automata* (dfa) defined over a one letter input alphabet $\Sigma = \{1\}$. A dfa will be denoted as a 4-tuple $A = (Q, \delta, q_0, F)$, with the usual meaning (see, e.g., [4]).

It is not difficult to observe that the transition graph of a unary dfa consists of a path, which starts from the initial state, followed by a cycle of one or more states. All automata we will consider are complete and connected. As in [2], the *size* of a dfa A is the pair (λ, μ), where $\lambda \geq 1$ and $\mu \geq 0$ denote the number of states in the cycle and in the path, respectively. Throughout the paper, we will use the following convention to denote any unary automaton $A = (Q, \delta, q_0, F)$ of size (λ, μ): the set of states is denoted as $Q = \{q_0, q_1, \ldots, q_{\mu-1}, p_0, p_1, \ldots, p_{\lambda-1}\}$, where $q_0, q_1, \ldots, q_{\mu-1}$ are the states on the path, and $p_0, p_1, \ldots, p_{\lambda-1}$ are the states on the cycle (with $q_0 = p_0$ when $\mu = 0$); then $\delta(q_i, 1) = q_{i+1}$, for $i = 0, \ldots, \mu - 2$, $\delta(q_{\mu-1}, 1) = p_0$, and $\delta(p_i, 1) = p_{(i+1) \bmod \lambda}$, for $i = 0, \ldots, \lambda - 1$. A unary dfa of size (λ, μ) is represented in Figure 1.

Observing the form of unary dfa's, it is not difficult to conclude that unary regular languages correspond to *ultimately periodic sets* of integers. More precisely:

Theorem 1. *Given a unary language L and two integers $\lambda \geq 1$, $\mu \geq 0$, the following statements are equivalent:*

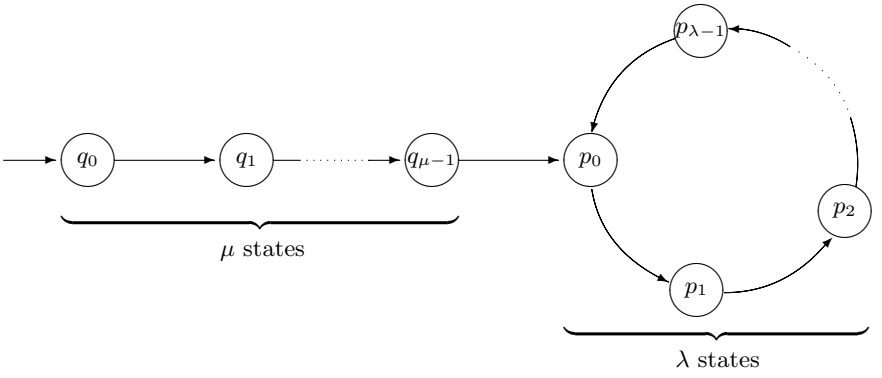


Fig. 1. A unary dfa of size (λ, μ)

(i) L *is accepted by a dfa of size* (λ, μ)*;*
(ii) for any $n \geq \mu$, $1^n \in L$ *if and only if* $1^{n+\lambda} \in L$.

A unary dfa is said to be *cyclic* if and only if its graph is a cycle. Languages accepted by cyclic automata are said to be *cyclic languages*. In other words, a unary language is cyclic if and only if it can be accepted by a dfa of size $(\lambda, 0)$, for some $\lambda \geq 1$. To emphasize the periodicity of L, we say that L is λ*–cyclic*. The following property can be easily proved by using Lemma 1:

Theorem 2. *If a unary language* L *is both* λ'*–cyclic and* λ''*–cyclic, for some* $\lambda', \lambda'' \geq 1$*, then* L *is* $\gcd(\lambda', \lambda'')$*–cyclic, too.*

In order to show the optimality of the constructions we will give, we now present a condition which characterizes minimal unary dfa's (see also [8, Lemma 1]):

Theorem 3. *A unary dfa* $A = (Q, \delta, q_0, F)$ *of size* (λ, μ) *is minimum if and only if both the following conditions are satisfied:*

(i) for any maximal proper divisor d *of* λ *(i.e.,* $\lambda = \alpha \cdot d$*, for some prime number* $\alpha > 1$*) there exists an integer* h*, with* $0 \leq h < \lambda$*, such that* $p_h \in F$ *if and only if* $p_{(h+d) \bmod \lambda} \notin F$*, i.e.,* $1^{\mu+h} \in L$ *if and only if* $1^{\mu+h+d} \notin L$*;*
(ii) $q_{\mu-1} \in F$ *if and only if* $p_{\lambda-1} \notin F$*, i.e.,* $1^{\mu-1} \in L$ *if and only if* $1^{\mu+\lambda-1} \notin L$.

Example 1. In order to prove the optimality of several results presented in the following sections, we will make use of the language $L = 1^{\mu+\lambda-1}(1^\lambda)^*$, where $\mu \geq 0$ and $\lambda \geq 1$. Using Theorem 3, it is easy to prove that the size of the minimum dfa A accepting L is (λ, μ). In particular, the only final state of A is $p_{\lambda-1}$.

The following result, which gives a tight evaluation of the state complexity of the union and intersection of unary regular languages, was recently proved by Shallit [11]:

Theorem 4. *Let L' and L'' be two languages accepted by unary automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively. The intersection (the union, respectively) of L' and L'' is accepted by a dfa of size $(\mathrm{lcm}(\lambda', \lambda''), \max(\mu', \mu''))$. Furthermore, for any $\lambda', \lambda'' \geq 1$, $\mu', \mu'' \geq 0$, there exists a pair of languages L', L'' witnessing the optimality of this bound.*

3 Concatenation

Given two dfa's with m and n states, accepting two languages L' and L'', respectively, the concatenation $L = L'L''$ is accepted by a dfa with $m2^n - 2^{n-1}$ states [13]. This result cannot be improved if the input alphabet contains at least three symbols. However, in the unary case, the number of states which are sufficient to recognize $L'L''$ reduces to mn. This number is also necessary, in the worst case, when m and n are relatively prime. As for the intersection, asymptotically, the worst case state complexity remains the same, even when m and n are not required to be relatively prime [8].

In this section, we further analyze the state complexity of the concatenation in the unary case, by evaluating the optimal size of an automaton accepting the concatenation of the languages accepted by two given unary dfa's. Moreover, we are able to show that, for some subclasses of unary regular languages, the size of the resulting automaton can be further reduced.

We start by considering two particular cases:

Theorem 5. *Given $\lambda', \lambda'' \geq 1$, $\mu', \mu'' \geq 0$, let L' and L'' be unary languages accepted by two dfa's A' and A'' of size (λ', μ') and (λ'', μ''), respectively.*

(i) If L'' is finite then $L'L''$ is accepted by a dfa of size $(\lambda', \mu' + \mu'' - 1)$.
(ii) If both languages L' and L'' are cyclic then $L'L''$ is accepted by a dfa of size $(\gcd(\lambda', \lambda''), \mathrm{lcm}(\lambda', \lambda'') - 1)$.

Proof. If L'' is finite, then any string in L'' has length less than μ''. Thus, given an integer $n \geq \mu' + \mu'' - 1$ such that $1^n \in L'L''$, there are two integers x and y such that $n = x+y$, $1^x \in L'$, $1^y \in L''$, $y < \mu''$, and $x \geq \mu'$. Since L' is accepted by a dfa of size (λ', μ'), this implies that even $1^{x+\lambda'} \in L'$. Hence, $1^{n+\lambda'} \in L'L''$. By similar arguments, we can also prove that, for any $n \geq \mu' + \mu'' - 1$, $1^{n+\lambda'} \in L'L''$ implies that $1^n \in L'L''$. Thus, in the light of Theorem 1, we conclude that $L'L''$ is accepted by an automaton of size $(\lambda', \mu' + \mu'' - 1)$.

If both L' and L'' are cyclic, then there exist two sets $Z' \subseteq \{0, \ldots, \lambda' - 1\}$, $Z'' \subseteq \{0, \ldots, \lambda'' - 1\}$, such that

$$L' = \{1^{z'+\lambda' k} \mid z' \in Z' \text{ and } k \geq 0\} \text{ and } L'' = \{1^{z''+\lambda'' j} \mid z'' \in Z'' \text{ and } j \geq 0\}.$$

In the light of Theorem 1, in order to prove that $L'L''$ is accepted by a dfa of size $(\gcd(\lambda', \lambda''), \mathrm{lcm}(\lambda', \lambda'') - 1)$, it is enough to show that, for any integer $z \geq \mathrm{lcm}(\lambda', \lambda'') - 1$, $1^z \in L'L''$ if and only if $1^{z+\gcd(\lambda', \lambda'')} \in L'L''$.

To this aim, consider an integer $w \geq \mathrm{lcm}(\lambda', \lambda'') - 1$ with $1^w \in L'L''$, and integers $z' \in Z'$, $z'' \in Z''$, $i, j \geq 0$, such that $w = z' + z'' + \lambda' i + \lambda'' j$. $\lambda' i + \lambda'' j$ is a multiple of $\gcd(\lambda', \lambda'')$, and it is greater than $\mathrm{lcm}(\lambda', \lambda'') - (\lambda' + \lambda'') + 1$. By Lemma 1, it turns out that even $\lambda' i + \lambda'' j + \gcd(\lambda', \lambda'')$ can be represented as $\lambda' x + \lambda'' y$, for some $x, y \geq 0$. Thus, it is easy to conclude that $1^{w+\gcd(\lambda', \lambda'')} \in L'L''$.

Using similar arguments, it is possible to show that $1^{w+\gcd(\lambda', \lambda'')} \in L'L''$ implies $1^w \in L'L''$. □

Now, we prove that the results stated in Theorem 5 are optimal. For the case of L'' finite, the following result can be easily proved:

Theorem 6. *Given some integers $\mu' \geq 0$, $\mu'', \lambda', \lambda'' \geq 1$, the infinite language $L' = 1^{\mu' + \lambda' - 1}(1^{\lambda'})^*$ and the finite language $L'' = 1^{\mu'' - 1}$ are accepted by two dfa's of size (λ', μ') and (λ'', μ''), respectively. Moreover, the size of the minimum dfa accepting the concatenation of L' and L'' is $(\lambda', \mu' + \mu'' - 1)$.*

The optimality of statement (ii) of Theorem 5 is a consequence (in the case $\mu' = \mu'' = 0$) of the following result:

Theorem 7. *Let $\mu', \mu'' \geq 0$, $\lambda', \lambda'' \geq 1$ be integer numbers. The languages*

$$L' = 1^{\mu' + \lambda' - 1}(1^{\lambda'})^* \text{ and } L'' = 1^{\mu'' + \lambda'' - 1}(1^{\lambda''})^*,$$

are accepted by two dfa's of size (λ', μ') and (λ'', μ''), respectively. Moreover, the size of the minimum dfa accepting $L'L''$ is $(\gcd(\lambda', \lambda''), \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1)$.

Proof. To see that languages L' and L'' are accepted by two dfa's of size (λ', μ') and (λ'', μ''), consider Example 1.

For the sake of simplicity, we will compute the size of the minimum dfa accepting $L'L''$ in the case $\mu' = \mu'' = 0$. The extension to the general case is trivial.

By Theorem 5(ii), there is an automaton A of size $(\gcd(\lambda', \lambda''), \mathrm{lcm}(\lambda', \lambda'') - 1)$, accepting $L = L'L'' = \{1^{\lambda' x + \lambda'' y - 2} \mid x, y \geq 1\}$. Using Theorem 3, we now show that A is minimum.

Since all numbers of the form $\lambda' x + \lambda'' y$, with $x, y \geq 0$, are multiple of $\gcd(\lambda', \lambda'')$, the difference between the lengths of two strings in L is a multiple of $\gcd(\lambda', \lambda'')$. Thus, the cycle of A, whose length is just $\gcd(\lambda', \lambda'')$, cannot contain more than one final state. This implies that condition (i) of Theorem 3 holds.

To show that even condition (ii) is satisfied, we now prove that $1^{\mathrm{lcm}(\lambda', \lambda'') - 2} \notin L$, while $1^{\mathrm{lcm}(\lambda', \lambda'') - 2 + \gcd(\lambda', \lambda'')} \in L$. To this aim, we consider numbers m of the form $m = \mathrm{lcm}(\lambda', \lambda'') - 2 + k \gcd(\lambda', \lambda'')$, with $k \geq 0$. The string 1^m belongs to L if and only if there are two integers $x, y \geq 1$ such that $m = \lambda' x + \lambda'' y - 2$, i.e., if and only if there are $x, y \geq 0$ such that $\lambda' x + \lambda'' y = \mathrm{lcm}(\lambda', \lambda'') - (\lambda' + \lambda'') + k \gcd(\lambda', \lambda'')$. By Lemma 1, this condition is satisfied if and only if $k \geq 1$. This implies that even condition (ii) of Theorem 3 holds. □

Now, we consider the general case:

Theorem 8. *Given $\mu', \mu'' \geq 0$, $\lambda', \lambda'' \geq 1$, let L' and L'' be unary languages accepted by two automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively. Then, the concatenation of L' and L'' is accepted by a dfa of size (λ, μ), where $\lambda = \mathrm{lcm}(\lambda', \lambda'')$ and $\mu = \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1$.*

Proof. (outline) Let X', X'' be the languages accepted by the initial paths of A', A'', i.e., $X' = L' \cap \{1^x \mid 0 \leq x < \mu'\}$, $X'' = L'' \cap \{1^x \mid 0 \leq x < \mu''\}$, and Y', Y'' the languages accepted by restricting automata A', A'' to their cyclic parts. Hence, $L' = X' \cup 1^{\mu'}Y'$ and $L'' = X'' \cup 1^{\mu''}Y''$. The product L of L' and L'' can be expressed as:

$$L = X'X'' \cup 1^{\mu'}X''Y' \cup 1^{\mu''}X'Y'' \cup 1^{\mu'+\mu''}Y'Y''. \tag{1}$$

Since the languages X', X'', Y', Y'', $1^{\mu'}$, and $1^{\mu''}$ can be accepted by dfa's of size $(1, \mu')$, $(1, \mu'')$, $(\lambda', 0)$, $(\lambda'', 0)$, $(1, \mu' + 1)$ and $(1, \mu'' + 1)$, respectively, using Theorem 4 and Theorem 5, it is not difficult to conclude that L is accepted by a dfa of size (λ, μ), where $\lambda = \mathrm{lcm}(\lambda', \lambda'')$ and $\mu = \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1$. □

We now study the optimality of the result stated in Theorem 8. First, we prove that this result is optimal when $\gcd(\lambda', \lambda'') > 1$. Subsequently, we will consider relatively prime λ' and λ'': we will show that in this case the number of states in the cyclic part can be further reduced.

Let us start by proving the following result:

Theorem 9. *For any $\mu', \mu'' \geq 2$, $\lambda', \lambda'' \geq 2$, such that $\gcd(\lambda', \lambda'') > 1$, there exists two unary languages L' and L'' which are accepted by two automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively, such that the concatenation of L' and L'' is accepted by a dfa of size (λ, μ), where $\lambda = \mathrm{lcm}(\lambda', \lambda'')$ and $\mu = \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1$.*

Proof. If λ'' divides λ' (λ' divides λ'', respectively), then the result is a consequence of Theorem 6. Now, suppose that λ' does not divide λ'' and λ'' does not divide λ', and consider the languages:

$$L' = 1^{\mu'+\lambda'-1}(1^{\lambda'})^* \cup 1^{\mu'-2} \text{ and } L'' = 1^{\mu''+\lambda''-1}(1^{\lambda''})^* \cup 1^{\mu''-2}.$$

It is not difficult to describe two automata A' and A'' of size (λ', μ') and (λ'', μ'') accepting L' and L'', respectively. From these automata, according to Theorem 8, an automaton A of size (λ, μ) accepting L can be obtained. We observe that a state p_x on the cycle of A, with $0 \leq x < \lambda$, is final if and only if there is an integer $k \geq 1$ such that either $x = kg - 1$, or $x = k\lambda' - 2$, or $x = k\lambda'' - 2$, where $g = \gcd(\lambda', \lambda'')$.

In order to show that A is minimum, we prove that both conditions *(i)* and *(ii)* of Theorem 3 are satisfied.

Consider a maximal proper divisor d of λ. Then, either λ' divides d, or λ'' divides d. Suppose that λ' divides d, i.e., $d = \beta\lambda'$, for some $\beta \geq 1$, and consider

$h = \lambda'' - 2$. Then $h + d = \lambda'' - 2 + \beta\lambda'$. So, $p_{(h+d) \bmod \lambda} \in F$ if and only if there exists an integer $k \geq 1$ such that either *(a)* $\lambda'' - 2 + \beta\lambda' = kg - 1$, or *(b)* $\lambda'' - 2 + \beta\lambda' = k\lambda' - 2$, or *(c)* $\lambda'' - 2 + \beta\lambda' = k\lambda'' - 2$.

Since g is greater than 1 and divides both λ' and λ'', the equality *(a)*, which reduces to $\lambda'' + \beta\lambda' = kg + 1$ cannot be verified, for any integer k. Also equality *(b)* cannot be verified since it implies that λ' divides λ''. Finally, equality *(c)* reduces to $\beta\lambda' = (k-1)\lambda''$; thus, it implies that $\beta\lambda'$, namely d, is a multiple of both λ' and λ'', i.e., a multiple of λ. This is a contradiction. Thus, we are able to conclude that $p_{(h+d) \bmod \lambda} \notin F$, while $p_h \in F$. The case of λ'' which divides λ' can be managed in a similar way. Hence, condition *(i)* of Theorem 3 is satisfied. Using Lemma 1, it is possible to verify that $1^{\mu'+\mu''+\mathrm{lcm}(\lambda',\lambda'')-2} \notin L$, while $1^{\mu'+\mu''+\mathrm{lcm}(\lambda',\lambda'')+\lambda-2} \in L$. Hence, even condition (ii) of Theorem 3 holds. This implies that A is minimum. □

Theorem 9 shows the optimality of the result stated in Theorem 8, for all $\mu', \mu'', \lambda', \lambda''$ such that $\gcd(\lambda', \lambda'') > 1$, with few exceptions for small μ', μ''.

We now consider the case of relatively prime λ' and λ''. The number of states in the cyclic part of the minimum dfa accepting the product of L' and L'' is less than $\mathrm{lcm}(\lambda', \lambda'')$. In particular, if both languages are infinite, then this number reduces to 1, while if L'' is finite it reduces to λ':

Theorem 10. *Let L' and L'' be unary languages accepted by two automata A' and A'' of size (λ', μ'), (λ'', μ''), respectively, with $\mu', \mu'' \geq 0$, $\lambda', \lambda'' \geq 1$, such that* $\gcd(\lambda', \lambda'') = 1$.

If both L' and L'' are infinite, then their concatenation is accepted by an automaton A of size $(1, \mu' + \mu'' + \lambda'\lambda'' - 1)$; if L'' is finite, then the concatenation of L' and L'' is accepted by an automaton of size $(\lambda', \mu' + \mu'' - 1)$.

These results are optimal, with the only exception of the trivial case $L'' = \emptyset$.

Proof. Suppose that both L' and L'' are infinite. The concatenation L of L' and L'' can be expressed as in equality (1) (Proof of Theorem 8). Since $\gcd(\lambda', \lambda'') = 1$, any string 1^x with $x \geq \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1$ belongs to $1^{\mu'+\mu''}Y'Y''$ and then to L. Hence, it is possible to conclude that a cycle of length 1 is sufficient. The optimality is a consequence of Theorem 7.

When L'' is finite, the result is an immediate consequence of Theorem 5 and of Theorem 6. □

Some Particular Cases

In the proof of Theorem 8, we have outlined the construction of an automaton A accepting the concatenation L of two languages L' and L'' accepted by two given unary dfa's A' and A''. To get the automaton A, in equality (1) we have expressed the language L as the union of some languages which are obtained by combining the cyclic and the noncyclic parts of L' and L''. When one or both noncyclic parts are empty, some of the languages on the right side of (1) are empty. Thus, evaluating in these cases the size of the resulting automata, one can easily get the following result:

Theorem 11. *Let L' and L'' be unary languages accepted by two automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively. The concatenation of L' and L'' is accepted by a dfa of size (λ, μ), where $\mu = \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1$ and:*

(i) if the initial path of A' does not contain any final state, then λ can be taken equal to λ';
(ii) if the initial path of A'' does not contain any final state, then λ can be taken equal to λ'';
(iii) if both the initial paths do not contain any final state, then λ can be taken equal to $\gcd(\lambda', \lambda'')$.

We point out that even the results stated in Theorem 11 are optimal when $\gcd(\lambda', \lambda'') > 1$. For statement (iii), this is a consequence of Theorem 7. This also implies the optimality of (ii) when $\lambda'' = \gcd(\lambda', \lambda'') = 2$. On the other hand, for $\lambda'' > 2$, the optimality of (ii) is given by the following result, whose proof is similar to that of Theorem 9:

Theorem 12. *Given $\mu', \lambda' \geq 2$, $\lambda'' > 2$, $\mu'' \geq 0$ such that $\gcd(\lambda', \lambda'') > 1$, consider the languages*

$$L' = 1^{\mu'+\lambda'-1}(1^{\lambda'})^* \cup 1^{\mu'-2} \quad L'' = 1^{\mu''+\lambda''-1}(1^{\lambda''})^*.$$

The languages L' and L'' can be accepted by two automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively. Furthermore, the minimum dfa accepting the concatenation $L = L'L''$ has size $(\lambda'', \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1)$.

We point out that the optimality of Theorem 11*(i)*, for $\gcd(\lambda', \lambda'') > 1$, can be shown in a similar way.

We conclude this section by summarizing, in Table 1 and in Table 2, the results we have proved concerning the state complexity of the concatenation of two languages L' and L'' accepted by two automata A' and A'' of size (λ', μ') and (λ'', μ''), respectively.

Table 1. State complexity of the concatenation, *when* $\gcd(\lambda', \lambda'') > 1$. X' and X'' denote the languages accepted by the initial paths of A' and A'', respectively, i.e, $X' = \{a^x \in L' \mid x < \mu'\}$ and $X'' = \{a^x \in L'' \mid x < \mu''\}$.

	$X'' \neq \emptyset$	$X'' = \emptyset$
$X' \neq \emptyset$	$(\mathrm{lcm}(\lambda', \lambda''), \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1)$ upper bound: Th. 8 lower bound: Th. 9	$(\lambda'', \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1)$ upper bound: Th. 11 lower bound: Th. 7 and Th. 12
$X' = \emptyset$	$(\lambda', \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1)$ upper bound: Th. 11 lower bound: Th. 7 and Th. 12	$(\gcd(\lambda', \lambda''), \mu' + \mu'' + \mathrm{lcm}(\lambda', \lambda'') - 1)$ upper bound: Th. 11 lower bound: Th. 7

Table 2. State complexity of the concatenation, *when* $\gcd(\lambda', \lambda'') = 1$.

	$\#L' = \infty$	$\#L' < \infty$
$\#L'' = \infty$	$(1, \mu' + \mu'' + \lambda'\lambda'' - 1)$ Th. 10	$(\lambda'', \mu' + \mu'' - 1)$ upper bound: Th. 5 lower bound: Th. 6
$\#L'' < \infty$	$(\lambda', \mu' + \mu'' - 1)$ upper bound: Th. 5 lower bound: Th. 6	$(1, \mu' + \mu'' - 1)$ upper bound: Th. 5 lower bound: trivial

4 Star

We conclude the paper by presenting, in this section, some considerations concerning the state complexity of the star operation in the unary case.

First of all, we recall the following result [13, Th. 5.3]:

Theorem 13. *If L is a unary regular language accepted by an n–state dfa, then L^* is accepted by a dfa with $(n-1)^2+1$ states. Furthermore, for any $n \geq 1$ this result cannot be improved.*

We observe that if L is accepted by an automaton of size (λ, μ), then the cycle in the minimum dfa accepting L^* cannot have more than λ states, i.e., the size (λ^*, μ^*) of the minimum automaton accepting L^* verifies $\lambda^* \leq \lambda$.

We now analyze some limit situations.

First, we suppose that $\mu = 0$, i.e., L is λ–cyclic. If $L = (1^\lambda)^*$, then $L = L^*$ and $(\lambda^*, \mu^*) = (\lambda, 0)$. Otherwise, let k be an integer such that $1^k \in L$ and $0 < k < \lambda$. Any string having length of the form $kx + \lambda y$, with $x \geq 1$ and $y \geq 0$, or $x = y = 0$, belongs to L^*. Hence, the length of the loop is $\lambda^* \leq \gcd(\lambda, k) \leq k$. In particular, when $L = 1^k(1^\lambda)^*$, by using Lemma 1, it is not difficult to conclude that $\lambda^* = \gcd(\lambda, k)$ and $\mu^* = \mathrm{lcm}(k, \lambda) - \lambda + 1$. For $k = \lambda - 1$ this reduces to $\lambda^* = 1$ and $\mu^* = (\lambda - 1)^2$, which exactly matches the number of states given in Theorem 13.

Now, we suppose that $\mu > 0$ and $\lambda = 1$. If $p_0 \in F$ then all strings of length greater than $\mu - 1$ belong to L. Thus L^* is accepted by an automaton of size $(1, \mu^*)$, with $\mu^* \leq \mu$. On the other hand, if $p_0 \notin F$ then L is finite. This case was analyzed in [1], proving that L^* is accepted by an automaton with at most $n^2 - 7n + 13$ states, where $n = \mu + \lambda = \mu + 1$ (this result, which is optimal, holds for $n > 4$ and for $n = 3$). We can suppose that $1^{\mu-1} \in L$, otherwise the size of the dfa accepting L can be reduced. If $L = \{1^{\mu-1}\}$, then L^* is accepted by an automaton of size $(\mu - 1, 0)$. If $L = \{1^s, 1^{\mu-1}\}$, with $0 \leq s < \mu - 1$, then, using Lemma 1, it can be shown that L^* is accepted by an automaton of size (λ^*, μ^*), with $\lambda^* = \gcd(\mu - 1, s)$ and $\mu^* = \mathrm{lcm}(\mu - 1, s) - \mu - s + 2$. In particular, when $s = \mu - 2$, we get $\lambda^* = 1$ and $\mu^* = \mu^2 - 5\mu + 6$ (note that for $n = \lambda + \mu$, $\lambda^* + \mu^*$ is exactly $n^2 - 7n + 13$). As pointed out in [1], the reader can verify that the size obtained in the last case is an upper limit for the case of L containing three or more words.

References

1. C. Câmpeanu, K. Culik II, K. Salomaa, and S. Yu. State complexity of basic operations on finite languages. Proceeding of the *Fourth International Workshop on Implementing Automata WIA'99*, 1999.
2. M. Chrobak. Finite automata and unary languages. *Theoretical Computer Science*, 47:149–158, 1986.
3. V. Geffert. Tally version of the Savitch and Immerman–Szelepcsényi theorems for sublogarithmic space. *SIAM J. Computing*, 22:102–113, 1993.
4. J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison–Wesley, Reading, MA, 1979.
5. J. Kaņeps. Regularity of one-letter languages acceptable by 2-way finite probabilistic automata. In *Fundamentals of Computation Theory (FCT'91)*, Lecture Notes in Computer Science 529, pages 287–296, 1991.
6. U. Liubicz. Bounds for the optimal determinization of nondeterministic autonomic automata. *Sibirskii Matemat. Journal*, 2:337–355, 1964. (In Russian).
7. C. Mereghetti and G. Pighizzini. Optimal simulations between unary automata. In *15th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1373, pages 139–149. Springer, 1998. To appear in *SIAM J. Computing.*
8. C. Nicaud. Average state complexity of operations on unary automata. In *Mathematical Foundations of Computer Science (MFCS'99)*, Lecture Notes in Computer Science 1672, pages 231–240, 1999.
9. C. Pomerance, J. Robson, and J. Shallit. Automaticity II: Descriptional complexity in the unary case. *Theoretical Computer Science*, 180:181–201, 1997.
10. J. Shallit and Y. Breitbart. Automaticity I: Properties of a measure of descriptional complexity. *Journal of Computer and System Sciences*, 53(1):10–25, 1996.
11. J. Shallit. State complexity and Jacobsthal's function. *Fifth International Conference on Implementation and Application of Automata, CIAA 2000.* These proceedings.
12. S. Yu. State complexity for regular languages. In *International Workshop on Descriptional Complexity of Automata, Grammars and Related Structures*, Preprint n. 17, pages 77–88. Department of Computer Science, Otto–von–Guericke University of Magdeburg, 1999.
13. S. Yu, Q. Zhuang, and K. Salomaa. The state complexities of some basic operations on regular languages. *Theoretical Computer Science*, 125(2):315–328, 1994.

Implementation of a Strategy Improvement Algorithm for Finite-State Parity Games

Dominik Schmitz and Jens Vöge*

Lehrstuhl für Informatik VII
RWTH Aachen, D-52056 Aachen, Germany
schmitz@i7.informatik.rwth-aachen.de
voege@informatik.rwth-aachen.de

Abstract. An implementation of a recent algorithm (Vöge, Jurdzinski, to appear in CAV 2000) for the construction of winning strategies in infinite games is presented. The games under consideration are "finite-state parity games", i.e. games over finite graphs where the winning condition is inherited from ω-automata with parity acceptance. The emphasis of the paper is the development of a user interface which supports the researcher in case studies for algorithms of ω-automata theory. Examples of such case studies are provided which might help in evaluating the (so far open) asymptotic runtime of the presented algorithm.

1 Introduction

The algorithmic approach to automata theory, which has been so successful in the theory of automata over finite words, has not yet produced results of similar strength in the theory of ω-automata, i.e., automata over infinite sequences. Although many intriguing and mathematically powerful constructions exist in ω-automata theory, the high complexity of most algorithms has so far prevented practical applications. Examples of such constructions are the Safra construction for determinizing ω-automata, the complementation of Büchi automata, and the construction of winning strategies in infinite games.

The package OMEGA of algorithms in ω-automata theory, presently under development at RWTH Aachen, should support the researcher in getting experience with the performance of these algorithms and in carrying out case studies. In the present paper we address a problem from ω-automata theory which belongs to the theory of infinite games: the construction of winning strategies in parity games. These games have a direct connection to the model-checking problem for the modal μ-calculus (cf. [3]). One of the central open questions in the field asks whether this model-checking problem or, equivalently, the construction of winning strategies in parity games is possible in polynomial time.

In [8] a new algorithm was developed, which has the potential of being more efficient than previously known procedures (e.g., [2,5,4]). The new algorithm

* Both authors are supported by the Deutsche Forschungsgemeinschaft (DFG), project Th 352/5-3.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 263–271, 2001.

follows the idea of strategy improvement as devised by Puri [6], but now in a completely discrete setting rather than over a dense value domain. The algorithm is hard to analyze; indeed, it is open whether the running time of this concrete algorithm can be bounded by a polynomial. Furthermore, the execution of the algorithm is hard to follow since it induces in each step a global change over the given game graph. In this situation, it is helpful to study the algorithm with an implementation and some auxiliary functions which help in assessing the main characteristics of its behaviour.

In the present paper we report on this implementation work, and on the user interface, and we discuss the output of the program in some simple example cases. In this way, we prepare a platform also for model-checking in the full μ-calculus (this issue is, however, not treated in the present work).

The paper is structured as follows: We present, in section 2, a short outline of the algorithm (for more detailed pseudocode see [8]) and describe some improvements which make an implementation more efficient. Based on these considerations, we have implemented the algorithm in C.

The focus of this paper is, however, the user's view of the program and the discussion of some case studies. These issues are treated in sections 3 and 4.

In section 3, we first present a Lisp–based input language for describing large game graphs. Secondly, the output format is discussed, including some features which support the user in analyzing the performance of the algorithm. Three essential parameters are isolated: The number of iterations of the main loop of the algorithm, the number of changes of strategy for each vertex, and (regarding the global running time) the number of edge traversals in the search procedures of the algorithm. For the second parameter a two-dimensional output (in matrix format) is developed.

Section 4 illustrates the use of the program. From the case studies considered so far we only discuss here an example taken from [1], where a classical (exponential-time) solution of parity games was given.

2 Parity Games and the Construction of Winning Strategies

A parity game is an infinite game played on a finite colored graph (the "arena" of the game). The game graph is of the form $G = (V_0, V_1, E, c)$. The set V of vertices is the union of the two disjoint sets vertex V_0 and V_1, and we have $E \subseteq (V_0 \times V_1) \cup (V_1 \times V_0)$, i.e., an edge leads from a V_0-vertex to a V_1-vertex or vice versa. We assume $vE \neq \emptyset$ for all vertices v.

A play starting from vertex v is an infinite sequence $v_0 v_1 v_2 \ldots$ of vertices with $(v_i, v_{i+1}) \in E$ for all $i \geq 0$; one imagines players 0 and 1 moving a token along edges through the graph in a nonterminating sequence of steps. If $v_i \in V_0$ it is the turn of player 0 to decide for an edge leaving v_i, otherwise player 1 has to choose an edge leaving v_i.

The winner of a play $v_0 v_1 v_2 \ldots$ is defined with a reference to the component c of the game graph, which is a function (called coloring) $c : V \to \{0, \ldots, k\}$

for some k. The play $v_0v_1v_2\ldots$ is won by player 0 if in the associated sequence $c(v_0)c(v_1)c(v_2)\ldots$ of colors the maximal color occurring infinitely often is even; otherwise player 1 wins the play.

It is a well-known theorem of the theory of infinite games (see e.g. [2,7]) that parity games (even over infinite game graphs) are "determined" in the following strong sense: From each vertex either player 0 or player 1 can force a win by a *positional winning strategy*. More precisely, the vertex set is partitioned in two *winning regions* W_0, W_1 such that player 0 wins starting from any vertex in W_0 by a fixed choice of outgoing edges for the V_0-vertices in W_0, and analogously player 1 wins starting from any vertex in W_1 by a fixed choice of outgoing edges for the V_1-vertices in W_1. The "solution of the game" is given by the sets W_0, W_1 and corresponding edge sets $E_0 \subseteq (V_0 \cap W_0) \times V_1$ and $E_1 \subseteq (V_1 \cap W_1) \times V_0$. As shown by [3], the problem of solving parity games over finite graphs is in NP $\cap$ co-NP and polynomially equivalent to the model-checking problem for the full modal μ-calculus. All known algorithms [2,5,4] have an exponential worst case running time. A key problem of the theory of program verification is to settle the question whether parity games can be solved in polynomial time.

In [8], a new algorithm was developed, based on the idea of strategy improvement (which originates in work on stochastic games and mean pay-off games). The running time of this algorithm can be trivially bounded by the number of possible strategies (say of player 0), which gives an exponential upper bound, but it is an open question whether this can be sharpened to a polynomial bound. This motivates an experimental study of the algorithm as initiated in this paper.

Let us sketch the algorithm of [8]. The key idea is a discrete valuation of plays and a process which successively modifies strategies of player 0 to guarantee higher and higher values of associated plays. To define this valuation of plays, we use a "relevance order" $<$ of the vertex set, which is a total order refining the order according to the coloring (so higher colors signal higher relevance). The vertices of the highest even color are the most valuable for player 0, while the vertices of the highest odd color are most valuable for the opponent, player 1. This defines the "reward ordering" $\prec$ (for player 0), obtained by concatenating the reversed relevance ordering of the odd-colored vertices and the relevance ordering of the even-colored vertices. (So a higher position of a vertex in the reward ordering signals an advantage for player 0.)

Assume a strategy pair (σ, τ) of the two players is given, where σ is defined by the choice of one out-edge for each vertex in V_0, similarly τ by the choice of one out-edge for each vertex in V_1. The strategy pair induces a well-defined play once an initial vertex is given; moreover, this play will end in a loop (which is completed when the first repetition of a vertex occurs). Given (σ, τ), each vertex v will get a value (also called *play profile*) which is extracted from the associated play starting in v. Given this valuation on the whole game graph (and an ordering of the value set), a strategy improvement step of player 0 will then consist in locally redefining the choice of his out-edges: he will pick, for each vertex u, an edge to a neighbour vertex v of highest value.

The play profiles are composed of three components, referring to the relevance order introduced above. Recall that for a given strategy pair (σ, τ) and vertex v the resulting play will end in a loop L. The three components r, P, m of the play profile of v are, in their order of significance, defined as follows: The first component is the most relevant vertex r of L, and called positive if its color is even, otherwise negative. Similarly, a play profile (r, P, m) is called positive (resp. negative) iff r is. We call r shortly "the most relevant loop vertex associated to v". The second component P of the play profile is the set of vertices more relevant than r and visited before entering the loop L (from v), and the third component is the number m of vertices visited before reaching r.

The play profiles are linearly ordered by an ordering $\prec$ as follows:

$$(r, P, m) \prec (r', P', m') \iff \begin{cases} r \prec r' \\ \vee\ (r = r' \wedge P \prec P') \\ \vee\ (r = r' \wedge P = P' \wedge r' \in V_- \wedge m < m') \\ \vee\ (r = r' \wedge P = P' \wedge r' \in V_+ \wedge m > m') \end{cases}$$

where $P \prec P'$ holds if there is a most relevant vertex in the symmetric difference of P and P' that is either positive and in P' or negative and in P.

The algorithm is outlined in Figure 1.

Strategy Improvement Algorithm

1. Choose an arbitrary strategy σ for player 0 (by picking an out-edge for each vertex in V_0).
2. Evaluate this strategy for player 0 (by computing an "optimal response" strategy τ of player 1, which induces together with σ a valuation of the game graph with play profiles).
3. By picking a neighbour vertex of highest value (play profile) for each vertex in V_0, construct a new strategy σ for player 0 and continue with step 2; if the new strategy σ was assumed already before, proceed with step 4.
4. The winning region W_0 (resp. W_1) consists of the vertices with positive (resp. negative) play profile, and the desired winning strategies of player 0 and 1 are extracted from the computed strategies σ and τ on the sets W_0 and W_1.

Fig. 1. Strategy Improvement Algorithm

The crucial step in this algorithm is the computation of the "optimal response strategy" of player 1 in step 2. For these details we have to refer the reader to the paper [8]. In our implementation, written in C, some algorithmic improvements over [8] are realized, which significantly affect the running time. For the sake of completeness we list them here (but skip the algorithmic justification):

1. Only play profiles of vertices v are computed where the associated most relevant loop vertex r is negative.
2. A strategy σ is not changed at vertex v when the associated most relevant loop vertex is positive.
3. The second component of the play profile is represented in tree form, which gives a linear memory bound for these components over all vertices.

In the sequel, we concentrate on the second component of the implementation, a user interface written in Common Lisp. This user interface allows to name vertices by arbitrary symbols or tuples of symbols. Using the interactive environment of Lisp, one can easily generate scalable game graphs, as needed for parametrized case studies.

3 User Interface

3.1 Input Language

The input language is embedded in Common Lisp, extended by some functions and macros that allow a convenient description of parity game graphs.

The synthesis algorithm can be called from within a Common Lisp environment by

(strategy vertices−of−player0 vertices−of−player1 edge−list colors order)

This function call starts with its name *strategy*. The parameters *vertices−of−player0* and *vertices−of−player1* are the lists of vertices associated with player 0 resp. 1. The parameter *edge−list* is a list of edges; each edge is itself a list of the form *(source target)* where *source* and *target* are vertices. The parameter *colors* is a list *(color0 color1 ... colorn)* where *colori* is the list of vertices of color i, usually in the order of increasing relevance.

The parameter *order* can be omitted; in this case (which corresponds to the option *: last−best*) the colors are considered to be ordered by increasing relevance. If the parameter *order* is set to be *: first−best*, then the colors are considered to be listed in decreasing relevance, as well as the vertices in each list *colori*.

A vertex can be any Lisp object. The equality of two vertices is determined by the function *equal*.

Example Consider the parity game graph $G_2 = (V_0, V_1, E, c)$ with $V_0 = \{2, 4\}$, $V_1 = \{1, 3\}$, $E = \{(1,2), (2,1), (2,3), (3,4), (4,3)\}$ and $c(1) = 1$, $c(2) = 1$, $c(3) = 1$, $c(4) = 0$ as presented in Figure 1. We indicate vertices in V_0 by circles and vertices in V_1 by boxes.

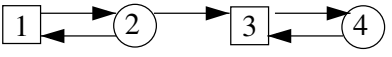


Fig. 2. Example graph G_2.

The procedure for computing the winning regions and corresponding winning strategies is called as follows:[1]

```
(strategy  '(2 4)
           '(1 3)
          '((1 2) (2 1) (2 3) (3 4) (4 3))
           '((4) (3 2 1))
           :first-best)
```

Instead of listing the elements for the parameters of such a function call explicitly, we shall use functions that generate the parameters for a call of *strategy*.

Macrolanguage For larger examples, an algorithmic definition of game graphs is appropriate. We develop a notation for such definitions, again in the form of a Lisp macro, now with the name *parity-game*. It provides an environment in which the three macros vertex, *edge* and *dedge* are available. These introduce the vertices, the edges, and for convenience also double edges ("dedges") which are considered present in both directions.

Example We shall explain its use for the example graphs G_n given in Figure 3.

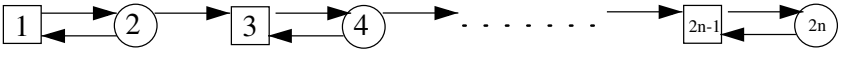


Fig. 3. Example graph G_n.

A specification in Pascal–like pseudo code is given below as well as the corresponding Lisp-Code.

```
for v:= 1 to n * 2 do
  v is vertex with:
    if even then v in V0
            else v in V1
    if v = 2 * n then color(v) = 0
                 else color(v) = 1
for i:= 1 to n do
  (2 * i - 1, 2 * i) is d-edge
for i:= 1 to n - 1 do
  (2 * i, 2 * i + 1) is edge
```

```
(parity-game
 (for (v 1 (* n 2))
   (vertex (v)
           (if (evenp v) 0 1)
           (if (= v (* 2 n)) 0 1)))

 (for (i 1 n)
   (dedge ((- (* 2 i) 1)) ((* 2 i)) ))
 (for (i 1 (- n 1))
   (edge ((* 2 i)) ((+ (* 2 i) 1)) )))
```

The introduction of the edges is self-explanatory. For the vertices, one has to declare, besides the name, the player to which it belongs, and its color (in the example the even vertices belong to player 0, the other to player 1, and the color is set 0 for vertex $2n$, otherwise 1). In this environment, a vertex is always a list. The conditionals are written as usual, with the if-clause and the two values for the yes- resp. no-case. We introduced also a for-loop with the obvious meaning.

[1] The quote symbol in front of a parameter tells the interpreter that it is constant, otherwise the following list would be also interpreted as a function call.

For the examples discussed below we also use definitions of the same format (not given in this extended abstract).

3.2 Output

The output contains the following data: the computed winning regions of players 0 and 1, optimal strategies for the two players, and the number of iterations used (improvement steps). The latter is the critical parameter for separating polynomial from exponential behaviour of the algorithm.

The listed strategies are winning on the respective winning region. On the complement the computed strategy is of course not winning, but just as good as possible with respect to the reward order (in the sense of section 2).

Example In the example considered above, one sees the following output (which says that player 0 wins from each vertex by choosing the listed edges from vertices in V_0):

```
(4 3 2 1)       ; winning region for 0
NIL             ; winning region for 1
((2 3) (4 3)) ; winning, resp. optimal strategy for 0
((1 2) (3 4)) ; winning, resp. optimal strategy for 1
2               ; number of iterations
```

For a more detailed analysis of the algorithm we found the following parameters to be significant (and accessible by separate calls of corresponding functions):

The overall running time is measured in the *number of edge traversals*. By this we mean the number of all edges traversed for computing the valuations. The number of executions of each other basic operation is bounded by the number of edge traversals. That means runtime is approximately linear in the number of edge traversals; and the number of edge traversals is a system independent performance measure.

The complicated feature of the algorithm is the possibility of changing the strategy for each vertex of player 0 in each step. An interesting overview of the execution is thus obtained by a matrix in which each row describes the result of a given iteration for the vertices of player 0. We call this matrix the *improvement trace*. Each vertex is associated to one column, in the order of increasing relevance. The i-th row records the result for the i-th iteration. A mark at position (i, j) of the matrix indicates that the strategy for vertex j (in the given order) is changed in iteration i. We distinguish two kinds of marks, star and dot. By this we record whether in the corresponding improvement step the most relevant loop vertex (see section 2 above) of vertex j is changed during iteration i or not. If this happens then the mark is taken to be a star, otherwise a dot.

An important observation in this connection is the following: If n is the number of vertices, there may by only n^2 number of iterations containing a star. Hence the crucial point in deciding whether the algorithm is polynomial time is

to bound the number of iterations without star entries. (In the present extended abstract we have to skip the examples which illustrate this.)

4 Remarks on Case Studies

The first case studies carried out with the system described above were concerned with example games discussed in previous literature on the subject, where different approaches to the problem of solving parity games were tried. Two such papers are [1], based on a recursive algorithm of McNaughton [5], and [4], where progress measures were applied. In both cases, suitable families of game graphs can be defined which prove exponential worst case behaviour for each algorithm. The analysis of the present algorithm with our implementation and user interface shows that here a linear number of iterations suffices to reach termination (which shows its polynomial time behaviour in these special cases).

In this extended abstract, we deal only with the case study of [1].

Let us describe the additional output function of the implementation for the version of the graphs of [1] with 136 vertices, where the vertex of player 0 are numbered $0, 2, \ldots, 134$. The numbers of vertices of player 0 are noted as names of vertices in the computed improvement trace (in decimal notation top-down).

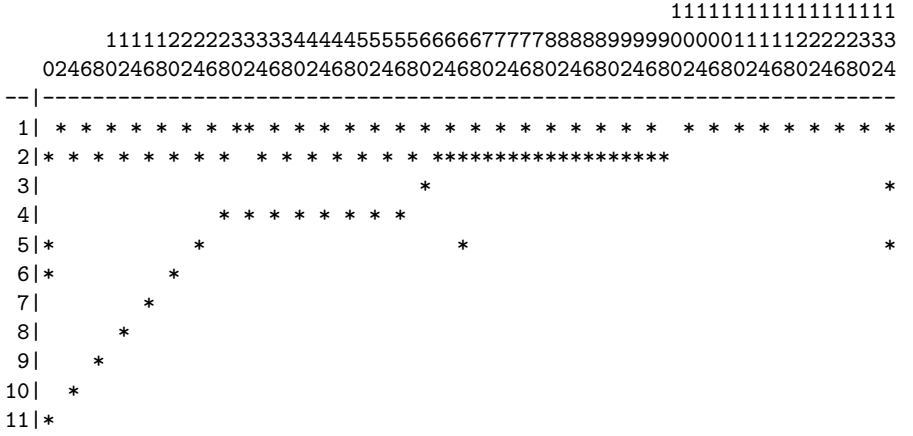


Fig. 4. Example of Improvement Trace

The output (see Figure 4) gives information about the iterations 1 to 11. From the improvement trace one infers that the algorithm adjusts several choices simultaneously until the 6th iteration and then proceeds by adjusting the strategy at the single vertices $16, 12, 8, 4, 0$ in the remaining iterations. In this case, there are 12 iterations, the last one offering no change in strategy (which causes termination) and hence suppressed in the trace.

5 Conclusion

We have provided an implementation of the algorithm of [8], extended it by a user interface, and applied it in several case studies. In ongoing work, more and larger examples are analyzed, and the program is connected to other procedures of the system OMEGA. Moreover, the algorithm is being applied to the model-checking problem of the modal μ-calculus, with a suitably extended user interface.

References

[1] N. Buhrke, H. Lescow, and J. Vöge. Strategy construction in infinite games with Streett and Rabin chain winning conditions. In T. Magaria and B. Steffen, editors, *TACAS'96*, volume 1055 of *LNCS*, pages 207–225. Springer, 1996.

[2] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symp. on the Foundations of Computing*, pages 368 – 377, 1991.

[3] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for fragments of μ-calculus. In C. Courcoubetis, editor, *Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 385–396, Berlin, 1993. Springer Verlag.

[4] M. Jurdziński. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 1770 of *LNCS*, pages 290–301, Lille, France, February 2000. Springer.

[5] R. McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl Logic*, 65:149 – 184, 1993.

[6] A. Puri. *Theory of hybrid systems and discrete event systems.* Ph.d. thesis, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, Dec. 1995. Memorandum No. UCB/ERL M95/113.

[7] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, New York, 1997.

[8] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification (CAV)*, LNCS. Springer Verlag, 2000.

State Complexity and Jacobsthal's Function

Jeffrey Shallit*

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
shallit@graceland.uwaterloo.ca

Abstract. We find bounds for the state complexity of the intersection of regular languages over an alphabet of one letter. There is an interesting connection to Jacobsthal's function from number theory.

1 Introduction

The *state complexity* of a regular language L, written $\mathrm{sc}(L)$, is the number of states in the smallest deterministic finite automaton (DFA) accepting L. Several papers, such as [12], address the question of obtaining good upper bounds on the state complexity of basic operations such as LL', L^*, etc., in terms of the state complexity of L and L'. Additional questions of interest arise when one restricts L, L' to be, for example, finite languages.

The standard product construction for automata (e.g., [3, pp. 59–60]) easily shows that if $\mathrm{sc}(L) = n$, $\mathrm{sc}(L') = n'$, then $\mathrm{sc}(L \cap L') \leq nn'$. This upper bound of nn' can actually be attained for all $n, n' \geq 1$ provided the underlying alphabet has at least two letters. Indeed, as Yu and Zhuang observe [11], we can let

$$L = \{x \in (a+b)^* \ : \ |x|_a = n\}$$

and

$$L' = \{x \in (a+b)^* \ : \ |x|_b = n'\},$$

where $|x|_c$ denotes the number of occurrences of the symbol c in the string x. A similar construction works for unary alphabets provided $\gcd(n, n') = 1$. However, determining the best upper bound for unary languages when $\gcd(n, n') > 1$ was stated as an open problem by Yu [10].

In this paper we prove bounds for the state complexity of the intersection of unary regular languages, and we show the problem is related to an interesting function from number theory due to Jacobsthal. Since $L \cup L' = \overline{\overline{L} \cap \overline{L'}}$, and $\mathrm{sc}(L) = \mathrm{sc}(\overline{L})$, our results apply equally well to the state complexity of the union of unary regular languages.

C. Nicaud [7] has recently investigated the *average* state complexity for various operations on unary languages, including intersection.

* Research supported in part by a grant from NSERC.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 272–278, 2001.

2 Unary Deterministic Finite Automata

Let $\Sigma = \{a\}$ and let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton with n states. Then by the pigeonhole principle, the transition diagram of M has a "tail" consisting of $t \geq 0$ states and a "cycle" of $c \geq 1$ states. Furthermore, if the transition diagram is connected (as we may assume without loss of generality) then $n = t + c$. See Figure 1.

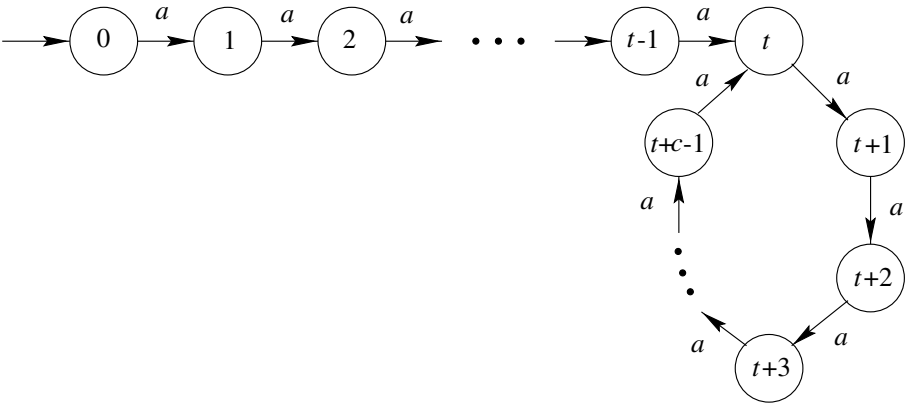


Fig. 1. Transition diagram of M (accepting states not identified)

We define $T(M) = t$ and $C(M) = c$. It follows that there exist sets $A \subseteq \{\epsilon, a, \ldots, a^{t-1}\}$ and $B \subseteq \{a^t, a^{t+1}, \ldots, a^{t+c-1}\}$ such that

$$L(M) = A + B(a^c)^* \tag{1}$$

Theorem 1. *Let M and M' be two unary deterministic finite automata. Suppose the transition diagram of M (resp. M') has a tail of size t and a cycle of size c (resp. t', c'). Then the state complexity of $L(M) \cap L(M')$ is $\leq \max(t, t') + \mathrm{lcm}(c, c')$.*

Proof. Write $L(M) = A + B(a^c)^*$, as in Eq. (1), and $L(M') = A' + B'(a^{c'})^*$. We have $a^s \in L(M)$ iff [$s < t$ implies $a^s \in A$ and $s \geq t$ implies there exists $a^y \in B$ such that $s \equiv y \pmod{c}$]. Similarly, $a^s \in L(M')$ iff [$s < t'$ implies $a^s \in A'$ and $s \geq t'$ implies there exists $a^z \in B'$ such that $s \equiv z \pmod{c'}$].

Then, by the Chinese remainder theorem, there exists a set B'' such that if $s \geq \max(t, t')$, then $a^s \in L(M) \cap L(M')$ iff there exists $u \in B''$ such that $s \equiv u \pmod{\mathrm{lcm}(c, c')}$. Hence we can accept $L(M) \cap L(M')$ using a cycle of size $\mathrm{lcm}(c, c')$ and a tail of size $\max(t, t')$. The upper bound follows. ■

We now show that the upper bound in Theorem 1 is best possible.

Theorem 2. *For all $t, t' \geq 0$ and $c, c' \geq 1$ there exist two deterministic finite automata M, M' with $T(M) = t$, $C(M) = c$, $T(M') = t'$, $C(M') = c'$ such that* $\mathrm{sc}(L(M) \cap L(M')) = \max(t, t') + \mathrm{lcm}(c, c')$.

Proof. Let $l = \mathrm{lcm}(c, c')$.

If $t = t' = 0$, then let $L = (a^c)^*$, $L' = (a^{c'})^*$. Then L (respectively L') may be accepted by a DFA M (respectively M') with $T(M) = 0$ and $C(M) = c$ (respectively $T(M') = 0$ and $C(M') = c'$). Now $L \cap L' = (a^l)^*$. It is now easy to see that $(a^l)^*$ may be accepted by a DFA with l states, and by the Myhill-Nerode theorem (e.g., [3, §3.4]), this is best possible.

Otherwise, at least one of t, t' is non-zero. Without loss of generality, assume $t \geq t'$ and hence $t > 0$. Define

$$L = a^{t+c-1}(a^c)^*$$
$$L' = a^r(a^{c'})^*$$

where $r := (t-1) \bmod c'$. It is easy to see that L (respectively, L') can be accepted by a DFA M (respectively, M') with $T(M) = t$, $C(M) = c$ (respectively, $T(M') = t'$, $C(M') = c'$). (In fact, L' can be accepted by a DFA M' with $T(M') = 0$.)

We claim $L \cap L' = a^{t+l-1}(a^l)^*$. To see this, note that $a^s \in L$ iff $s = (t + c - 1) + kc$ for some integer $k \geq 0$. Similarly, letting $t - 1 = qc' + r$ with $0 \leq r < c'$, we have $a^s \in L'$ iff $s = r + jc'$ for some integer $j \geq 0$, i.e., iff $s = (t-1) + (j-q)c'$. Thus $a^s \in L \cap L'$ iff $t + c - 1 + kc = (t-1) + (j-q)c'$, which is the case iff $(k+1)c = (j-q)c'$. But this equation has integer solutions iff $(k+1) = bc'/g$ and $j - q = bc/g$ for some integer b, where $g = \gcd(c, c')$. But $k \geq 0$ iff $b \geq 1$. It now follows that

$$\begin{aligned} L \cap L' &= \{a^{(t+c-1)+(bc'/g-1)c} \ : \ b \geq 1\} \\ &= \{a^{t-1+bl} \ : \ b \geq 1\} \\ &= a^{t+l-1}(a^l)^* \end{aligned}$$

as desired.

Now an easy application of the Myhill-Nerode theorem proves that

$$\mathrm{sc}(a^{t+l-1}(a^l)^*) = t + l.$$

■

After this paper was completed, I learned that Theorems 1 and 2 were obtained independently by G. Pighizzini [8].

Together, Theorems 1 and 2 imply that to understand the state complexity of the intersection of regular languages, we need to estimate the function

$$F(n, n') = \max_{\substack{1 \leq c \leq n \\ 1 \leq c' \leq n'}} (\max(n - c, n' - c') + \mathrm{lcm}(c, c')).$$

This in turn suggests studying the somewhat simpler and more natural function

$$G(n, n') = \max_{\substack{1 \le c \le n \\ 1 \le c' \le n'}} \operatorname{lcm}(c, c').$$

To the best of our knowledge, neither F nor G has been studied previously, although we will see in the next section that both functions are closely related to the Jacobsthal function.

3 Jacobsthal's Function

Jacobsthal's function $g(n)$ is defined to be the least integer r such that every set of r consecutive integers contains at least one integer relatively prime to n [5]. Below we show an interesting connection between this problem and state complexity for intersection of unary languages.

First, however, we state two known upper bounds on this function. The first is an explicit bound due to Kanold [6]:

Theorem 3. *Let $\omega(n)$ denote the number of distinct prime factors of the positive integer n. Then $g(n) \le 2^{\omega(n)}$ for all integers $n \ge 1$.*

The second bound is due to Iwaniec [4]:

Theorem 4. *There exists a constant c_1 such that $g(n) \le c_1(\log n)^2$ for all integers $n \ge 1$.*

First we obtain a lower bound on G (and hence F):

Theorem 5. *Let $n \le n'$. Then there exists a constant c_1 such that $F(n, n') \ge G(n, n') \ge nn' - c_1(\log n)^2 n$.*

Proof. By Iwaniec's theorem, there exists k with $0 \le k \le c_1(\log n)^2$ such that $\gcd(n, n' - k) = 1$. Hence $G(n, n') \ge n(n' - k) \ge n(n' - c_1(\log n)^2)$. ■

Carl Pomerance has kindly pointed out to me (personal communication) that the lower bound of Theorem 5 can be improved in the case where n and n' do not differ much in size, as follows. We use a result of Adhikari and Balasubramanian [1]:

Theorem 6. *If x, y are positive integers $\le N$, then there exist integers a, b with $a = O(\log\log\log N)$ and $b = O((\log N)/(\log\log N))$ such that $\gcd(x-a, y-b) = 1$.*

Using this theorem, we obtain the following:

Theorem 7.

(a) If $n \le n' \le \frac{n \log n}{(\log\log n)(\log\log\log n)}$, then there exists a constant c_2 such that $F(n, n') \ge G(n, n') \ge nn' - c_2 \frac{\log n}{\log\log n} n$.

(b) If $\frac{n \log n}{(\log \log n)(\log \log \log n)} \leq n' \leq \frac{n(\log n)^2}{\log \log \log n}$ *then there exists a constant* c_3 *such that* $F(n, n') \geq G(n, n') \geq nn' - c_3(\log \log \log n)n'$.

Next, we find a upper bound on F. First we prove the following lemma.

Lemma 1. *Let* n, n' *be fixed positive integers. The quantity*

$$Q(c, c') := \max(n - c, n' - c') + \operatorname{lcm}(c, c')$$

is maximized ($1 \leq c \leq n$, $1 \leq c' \leq n'$) *only if* $\gcd(c, c') = 1$.

Proof. Assume not. Then $Q(c, c')$ is maximized for some c, c' with $\gcd(c, c') = g > 1$. Assume without loss of generality that $n \geq n'$. For $n < 11$ the theorem can be verified by a simple computer program. Hence assume $n \geq 11$.

We have $\max(n-c, n'-c') < n$ and $\operatorname{lcm}(c, c') = \frac{cc'}{g} \leq \frac{n^2}{2}$, so $Q(c, c') < n + \frac{n^2}{2}$.

By Theorem 3 we know there exists a k, $1 \leq k \leq 2^{\omega(n)}$ such that $\gcd(n, n - k) = 1$. Since $Q(c, c')$ is a maximum, we have $Q(c, c') \geq n(n - k) + k > n(n - 2^{\omega(n)})$. Putting the inequalities for Q together, we get

$$n(n - 2^{\omega(n)}) < n + \frac{n^2}{2},$$

and so $n - 2^{\omega(n)} < 1 + \frac{n}{2}$. Thus $n < 2(2^{\omega(n)} + 1)$.

However, we claim that $n > 2(2^{\omega(n)} + 1)$ for $n \geq 11$. For $11 \leq n \leq 141$ this follows by an explicit calculation. Otherwise $n \geq 142$. We now use a theorem of Robin [9] which states $\omega(n) \leq t(n)$ where

$$t(n) := \frac{\log n}{\log \log n} + 1.45743 \frac{\log n}{(\log \log n)^2}.$$

Since $n \geq 142$, we have $\log \log n > 1.6$ and so

$$t(n) < \frac{\log_2 n}{1.6 \log_2 e} + 1.45743 \frac{\log_2 n}{2.56 \log_2 e} < .83 \log_2 n.$$

We thus obtain

$$2(2^{\omega(n)+1}) < 2(2^{t(n)} + 1) < 2(n^{.83} + 1)$$

and it is easily verified that $2(n^{.83} + 1) < n$ for $n \geq 70$. This contradiction completes the proof. ∎

Remark. Ming-wei Wang points out (personal communication) that a slightly weaker result is much easier to prove: namely, that Q achieves its maximum at some (c, c') with $\gcd(c, c') = 1$ (as opposed to "only if"). For if $\gcd(c, c') > 1$, then write $c = 2^{e_1} 3^{e_2} \cdots p_k^{e_k}$ and $c' = 2^{f_1} 3^{f_2} \cdots p_k^{f_k}$ where p_i is the k'th prime and p_k is the largest prime dividing either c or c'. Let $d = \prod_{\substack{1 \leq i \leq k \\ e_i \leq f_i}} p_i^{e_i}$ and $d' = \prod_{\substack{1 \leq i \leq k \\ e_i > f_i}} p_i^{f_i}$. Then $\operatorname{lcm}(c/d, c'/d') = \operatorname{lcm}(c, c')$, and hence we have $Q(c/d, c'/d') \geq Q(c, c')$. However $\gcd(c/d, c'/d') = 1$.

Remark. Note that $F(n,n') = \max_{\substack{1\le c\le n\\ 1\le c'\le n'}} \max(n-c, n'-c') + \operatorname{lcm}(c,c')$ does not necessarily achieve its maximum at the same pair (c,c') which maximizes $G(n,n') = \max_{\substack{1\le c\le n\\ 1\le c'\le n'}} \operatorname{lcm}(c,c')$. For example, $F(148,30) = 4295$, which is uniquely achieved at $(c,c') = (143,30)$, while $G(148,30) = 4292$, which is uniquely achieved at $(c,c') = (148,29)$.

We can now prove our upper bound.

Theorem 8. *There exist a constant c_4 and infinitely many distinct pairs n, n' with $n' < n$ such that $G(n,n') \le F(n,n') \le nn' - c_4\sqrt{\frac{\log n}{\log\log n}}\,n$.*

Proof. Let $d \ge 1$ be a fixed integer. Let $S_d = \{(i,j) : i,j \ge 0 \text{ and } i+j < d\}$. For each pair $(i,j) \in S_d$, choose a distinct prime $q_{i,j}$ from the set $\{p_1, p_2, \ldots, p_v\}$, where p_i denotes the i'th prime and $v = d(d+1)/2$. By the Chinese remainder theorem, we can find n, n' such that $q_{i,j} \mid n-i$ and $q_{i,j} \mid n'-j$ for all pairs $(i,j) \in S_d$. Furthermore, we may choose n and n' such that $K \le n' < 2K$, $2K \le n < 3K$, where $K := \prod_{1\le i\le v} p_i$. By the prime number theorem (e.g., [2]), we have $K = e^{(1+o(1))v\log v}$. Hence there exists a constant c_5 such that $d \ge c_5\sqrt{\frac{\log n}{\log\log n}}$.

It follows that $\gcd(n-i, n'-j) > 1$ for all pairs $(i,j) \in S_d$. By Lemma 1, we know that F cannot achieve its maximum when $(c,c') \in S_d$.

It follows that $F(n,n') \le \max_{b+c=d}((n-b)(n'-c)+d)$. But

$$\max_{b+c=d}((n-b)(n'-c)+d) \le nn' - dn' + d^2/4 + d.$$

Hence $F(n,n') \le n'(n-d) + d^2/4 + d$. Since $n' \ge n/3$, the desired result follows. ■

Remark. This result suggests defining a function $S(n)$ to be the least positive integer r such that there exists an integer m, $0 \le m \le r$, with $\gcd(r-i, m-j) > 1$ for $0 \le i,j < n$. By an argument similar to that given above, we know that $S(n) < e^{(1+o(1))2n^2\log n}$. The following table gives the first few values of $S(n)$:

n	$S(n)$	m
1	2	0
2	21	15
3	1310	1276

It is possible to prove through brute force calculation that $450000 < S(4) \le 172379781$. The upper bound follows from the fact that if

$$(x,y) = (172379781, 153132345),$$

then we have $\gcd(x-i, y-j) > 1$ for $0 \le i,j < 4$.

Acknowledgments. I thank Carl Pomerance and Ming-wei Wang for their helpful suggestions. I also thank Jean-Eric Pin for pointing out the work of Nicaud.

References

1. S. D. Adhikari and R. Balasubramanian. On a question regarding visibility of lattice points. *Mathematika*, 43:155–158, 1996.
2. E. Bach and J. Shallit. *Algorithmic Number Theory.* MIT Press, 1996.
3. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
4. H. Iwaniec. On the problem of Jacobsthal. *Dem. Math.*, 11:225–231, 1978.
5. E. Jacobsthal. Über Sequenzen ganzer Zahlen. von denen keine zu n teilerfremd ist. I-III. *Norske Vid. Selsk. Forh. Trondheim*, 33:117–139, 1960.
6. H.-J. Kanold. Über eine zahlentheoretische Funktion von Jacobsthal. *Math. Annalen*, 170:314–326, 1967.
7. C. Nicaud. Average state complexity of operations on unary automata. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *Proc. 24nd Symposium, Mathematical Foundations of Computer Science 1999*, volume 1672 of *Lecture Notes in Computer Science*, pages 231–240. Springer-Verlag, 1999.
8. G. Pighizzini. Unary language concatenation and its state complexity. *Proc. CIAA 2000.*
9. G. Robin. Estimation de la fonction de Tchebychef Θ sur le k-ième nombre premier et grandes valeurs de la fonction $\omega(n)$ nombre de diviseurs premiers de n. *Acta Arith.*, 42:367–389, 1983.
10. S. Yu. State complexity of regular languages. In *International Workshop on Descriptional Complexity of Automata, Grammars and Related Structures, Preproceedings*, pages 77–88. Department of Computer Science, Otto-von-Guericke University of Magdeburg, July 1999.
11. S. Yu and Q. Zhuang. On the state complexity of intersection of regular languages. *SIGACT News*, 22(3):52–54, Summer 1991.
12. S. Yu, Q. Zhuang, and K. Salomaa. The state complexity of some basic operations on regular languages. *Theoret. Comput. Sci.*, 125:315–328, 1994.

A Package for the Implementation of Block Codes as Finite Automata

Priti Shankar[1], K.Sasidharan[1], Vikas Aggarwal[1], and B.Sundar Rajan[2]

[1] Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India
[2] Department of Electrical Communication Engineering, Indian Institute of Science, Bangalore 560012, India
{priti,sasi,vikas}@csa.iisc.ernet.in, bsrajan@ece.iisc.ernet.in

Abstract. We have implemented a package that transforms concise algebraic descriptions of linear block codes into finite automata representations, and also generates decoders from such representations. The transformation takes a description of the code in the form of a $k \times n$ generator matrix over a field with q elements, representing a finite language containing q^k strings, and constructs a minimal automaton for the language from it, employing a well known algorithm. Next, from a decomposition of the minimal automaton into subautomata, it generates an overlayed automaton, and an efficient decoder for the code using a new algorithm. A simulator for the decoder on an additive white Gaussian noise channel is also generated. This simulator can be used to run test cases for specific codes for which an overlayed automaton is available. Experiments on the well known Golay code indicate that the new decoding algorithm is considerably more efficient than the traditional Viterbi algorithm run on the original automaton.

Keywords: block codes, minimal trellis, decoder complexity, subtrellis overlaying.

1 Introduction

The theory of finite state automata has many interesting connections to the field of error correcting codes [13]. After the early work on trellis representations of block codes[1,22,15,18,7], there has recently been a spate of research on minimal trellis representations of block error correcting codes[8,12,17]. Trellis descriptions are *combinatorial* descriptions, as opposed to the traditional *algebraic* descriptions of block codes. A minimal trellis for a linear block code is just the transition graph for the minimal finite state automaton which accepts the language consisting of the set of all codewords. With such a description, the decoding problem reduces to finding a cheapest accepting path in such an automaton(where transitions are assigned costs based on a channel model). However, trellises for many useful block codes are often too large to be of practical value. Of immense interest therefore, are *tail-biting* trellises for block codes, recently introduced in [3],

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 279–291, 2001.

which have reduced state complexity. The strings accepted by a finite state machine represented by a trellis are all of the same length, that is the *block length* of the code. Coding theorists therefore attach to all states that can be reached by strings of the same length l, a *time index* l. Conventional trellises use a linear time index, whereas tail-biting trellises use a circular time index. It has been observed[21] that the maximum state complexity of a tailbiting trellis at any time index can drop to the square root of the maximum state complexity of a conventional trellis for the code, thus increasing the potential practical applications of trellis representations for block codes. It is shown in [6] that tailbiting trellises can be viewed as *overlayed automata*, i.e. a superimposition of several identically structured finite state automata, so that states at certain time indices are shared by two or more automata. This view leads to a new decoding algorithm that is seen to be more efficient than the traditional Viterbi algorithm. Some preliminary details[10] are available for the construction of minimal tailbiting trellises from conventional trellises. The full theory is currently being worked out[11].

In this paper, we describe a software package that provides the following facilities.

1. It constructs a minimal finite state automaton(conventional trellis) for a linear block code from a concise algebraic description in terms of a generator matrix using the algorithm of Kschischang-Sorokine[12]. This involves two steps: First, the conversion of the generator matrix into *trellis oriented* form. This is a sequence of operations similar to Gaussian elimination. Second, the generation of the minimal trellis as the *product trellis* of smaller trellises. Note that the algebraic description is concise, consisting of a $k \times n$ matrix with entries from a field with q elements. The language consists of q^k strings.
2. Given a set of identically structured automata(subtrellises), that can be superimposed to produce an overlayed automaton(tailbiting trellis), it produces a decoder for the block code using a new algorithm described in [6].
3. Given the parameters of an additive white Gaussian noise(AWGN) channel it simulates the decoder on the overlayed automaton and outputs the decoded vector and other statistics for the range of signal to noise ratios(SNR) requested by the user.

Simulations on the hexacode[5], and on the practically important Golay code, the tailbiting trellises of which are both available[3], indicate that there is a significant gain in decoding rate using the new algorithm on the tailbiting trellis over the Viterbi algorithm on the conventional trellis.

We hope to augment the package with a module to convert from a minimal conventional trellis to a minimal tailbiting trellis when the technique(stated to be polynomial time in [10]) becomes available.

Section 2 describes conventional trellises for block codes and the Kschischang-Sorokine algorithm used to build the minimal trellis; section 3 defines tailbiting trellises and overlayed automata; section 4 describes the decoding algorithm; section 5 describes our implementation and presents some results obtained by running our decoder on a tailbiting trellis for the Golay code. Finally section 6 concludes the paper.

2 Minimal Trellises for Block Codes

In block coding, an *information sequence* of symbols over a finite alphabet is divided into *message blocks* of fixed length; each message block consists of k information symbols. If q is the size of the finite alphabet, there are a total of q^k distinct messages. Each message is encoded into a distinct *codeword* of n $(n > k)$ symbols. There are thus q^k codewords each of length n and this set forms a *block code* of length n. A block code is typically used to correct errors that occur in transmission over a communication channel. A subclass of block codes, the *linear block codes* has been used extensively for error correction. Traditionally such codes have been described *algebraically*, their algebraic properties playing a key role in *hard decision* decoding algorithms. In hard decision algorithms, the signals received at the output of the channel are *quantized* into one of the q possible transmitted values, and decoding is performed on a block of symbols of length n representing the received codeword, possibly corrupted by some errors. By contrast, *soft decision* decoding algorithms do not require quantization before decoding and are known to provide significant coding gains [4] when compared with hard decision decoding algorithms. That block codes have efficient *combinatorial* descriptions in the form of *trellises* was discovered in 1974 [1]. Other early seminal work in the area appears in [22] [15] [7] [18]. For background on the algebraic theory of block codes, readers are referred to the classic texts [14, 2,16]; for trellis structure of block codes, [19] is an excellent reference.
Let F_q be the field with q elements. It is customary to define linear codes algebraically as follows:

Definition 1. *A* linear block code C *of length* n *over a field* F_q *is a* k*-dimensional subspace of an* n*-dimensional vector space over the field* F_q *(such a code is called an* (n, k) *code).*

The most common algebraic representation of a linear block code is the generator matrix G. A $k \times n$ matrix G where the rows of G are linearly independent and which generate the subspace corresponding to C is called a *generator matrix* for C. Figure 1 shows a generator matrix for a $(4, 2)$ linear code over F_2. A

$$\mathbf{G} = \begin{bmatrix} 0\,1\,1\,0 \\ 1\,0\,0\,1 \end{bmatrix}$$

Fig. 1. Generator matrix for a $(4, 2)$ linear binary code

general block code also has a *combinatorial* description in the form of a *trellis*. We borrow from Kschischang and Sorokine [12] the definition of a trellis for a block code.

Definition 2. *A trellis for a block code C of length n, is an edge labeled directed graph with a distinguished* root *vertex s, having in-degree* 0 *and a distinguished* goal *vertex f having out-degree* 0*, with the following properties:*

1. *All vertices can be reached from the root.*
2. *The goal can be reached from all vertices.*
3. *The number of edges traversed in passing from the root to the goal along any path is n.*
4. *The set of n-tuples obtained by "reading off" the edge labels encountered in traversing all paths from the root to the goal is C.*

The length of a path (in edges) from the root to any vertex is unique and is sometimes called the *time index* of the vertex. One measure of the size of a trellis is the total number of vertices in the trellis. It is well known that minimal trellises for linear block codes are unique [18] and constructable from a generator matrix for the code [12]. Such trellises are known to be *biproper*. Biproperness is the terminology used by coding theorists to specify that the finite state automaton whose transition graph is the trellis, is deterministic, and so is the automaton obtained by reversing all the edges in the trellis. (Formal language theorists call such languages bideterministic languages). In contrast, minimal trellises for non-linear codes are, in general, neither unique, nor deterministic [12]. Figure 2 shows a trellis for the linear code in figure 1.

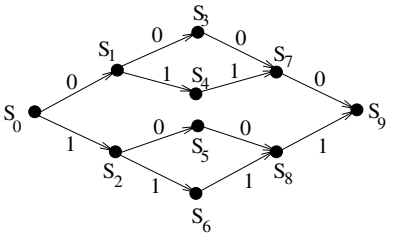


Fig. 2. A trellis for the linear block code of figure 1 with $S_0 = s$ and $S_9 = f$

2.1 Constructing Minimal Trellises for Block Codes

We briefly describe the algorithm given in [12] for constructing a minimal trellis,implemented in our package. An important component of the algorithm is the trellis product construction, whereby a trellis for a "sum" code can be obtained as a *product* of component trellises. Let T_1 and T_2 be the component trellises. We wish to construct the trellis product $T_1.T_2$. The set of vertices of the product trellis at each time index, is just the Cartesian product of the vertices of the component trellis. Thus if i is a time index, $V_i(T_1.T_2) = V_i(T_1) \times V_i(T_2)$. Consider $E_i(T_1) \times E_i(T_2)$, and interpret an element

$((v_1, \alpha_1, v_1'), (v_2, \alpha_2, v_2'))$ in this product, where v_i, v_i' are vertices and α_1, α_2 edge labels, as the edge $((v_1, v_2), \alpha_1 + \alpha_2, (v_1', v_2'))$ where + denotes addition in the field. If we define the i^{th} section as the set of edges connecting the vertices at time index i to those at time index $i+1$, then the edge count in the i^{th} section is the product of the edge counts in the i^{th} section of the individual trellises.

Before the product is constructed we put the matrix in *trellis oriented form* described now. Given a non zero codeword $C = (c_1, c_2, \ldots c_n)$, $start(C)$ is the smallest integer i such that c_i is non zero. Also $end(C)$ is the largest integer for which c_i is nonzero. The *span* of C is $[start(C), end(C)]$. By convention the span of the all 0 codeword $\mathbf{0}$ is []. The minimal trellis for the binary $(n, 1)$ code generated by a nonzero codeword with span $[a, b]$ is constructed as follows. There is only one path up to $a-1$ from index 0, and from b to n. From $a-1$ there are 2 outgoing branches diverging(corresponding to the 2 multiples of the codeword), and from $b-1$ to b, there are 2 branches converging. For a code over F_q there will be q outgoing branches and q converging branches. It is easy to see that this is the minimal trellis for the 1-dimensional code.
To generate the minimal trellis for C we first put the trellis into *trellis oriented form*, where for every pair of rows, with spans $[a_1, b_1], [a_2, b_2], a_1 \neq b_1$ and $a_2 \neq b_2$. We then construct individual trellises for the k 1-dimensional codes as described above, and then form the trellis product. Conversion of a generator matrix into trellis oriented form requires a sequence of operations similar to Gaussian elimination, applied twice. In the first phase, we apply the method to ensure that each row in the matrix starts its first nonzero entry at a time index one higher than the previous row. In the second phase we ensure that no two rows have their last nonzero entry at the same time index. We see that the generator matrix displayed earlier is already in trellis oriented form. The complexity of the Kschischang-Sorokine algorithm is $O(k.n + s)$ for an (n, k) linear code whose minimal trellis has s states.

3 Tailbiting Trellises

We borrow the definition of a tailbiting trellis from [10].

Definition 3. *A tailbiting trellis $T = (V, E, F_q)$ of depth n is an edge labelled directed graph with the following property. The vertex set can be partitioned as follows: $V = V_0 \cup V_1 \cup \ldots V_{n-1}$, such that every edge in T either begins at a vertex of V_i and ends at a vertex of V_{i+1} for some $i = 1, 2, \ldots n-2$ or begins at a vertex of V_{n-1} and ends at a vertex of V_0.*

The notion of a minimal tailbiting trellis is more complicated than that of a conventional trellis. The ordered sequence, $(|V_0|, |V_1|, \ldots |V_{n-1}|)$ is called the *state complexity profile* of the tailbiting trellis. For a given linear code C, the state complexity profiles of all tailbiting trellises for C form a partially ordered set under componentwise comparison. A trellis T is said to be smaller or equal to another trellis T', denoted by $T \leq_S T'$ if $|V_i| \leq |V_i'|$ for all $i = 0, 1, \ldots n-1$. It is

smaller if equality does not hold for all i in the expression above. We say that a tailbiting trellis is minimal under $\leq_S$ if a smaller trellis does not exist. For conventional trellises the minimal trellis is unique. However, for tailbiting trellises, there are, in general, several nonisomorphic minimal trellises that are incomparable with one another. Yet another ordering on tailbiting trellises is given by the *product ordering* $\leq_P$. This is a total ordering. $T <_P T'$ iff $\prod_{i=0}^{n-1} |V_i| < \prod_{i=0}^{n-1} |V_i'|$. It is stated in [10] that $T <_S T' \iff T <_P T'$. An outline for constructing a minimal tailbiting trellis for a linear block code is given in [10]. The complexity is stated to be $O(n^2)$ for a code of length n. The detailed theory in under preparation[11]. Figure 4 is a tailbiting trellis for the linear code of figure 1. Let $S_{max}(T)$ denote the maximum number of states of trellis T at any time index, when the index is allowed to range from 0 to n.
It is shown in [6] that a tailbiting trellis can be viewed as an *overlayed automaton*. This is a somewhat more natural view, we believe, and it also leads to an efficient decoding algorithm on tailbiting trellises.

3.1 Tailbiting Trellises as Overlayed Automata

An overlayed trellis has been introduced in [6], and we give the definition below. Let C be a linear code over a finite alphabet. Let $C_0, C_1, \ldots C_l$ be a partition of the code C, such that C_0 is a subgroup of C under the operation of componentwise addition over the structure that defines the alphabet set of the code(usually a field or a ring), and $C_1, \ldots C_l$ are cosets of C_0 in C. Let $C_i = C_0 + h_i$ where $h_i, 1 \leq h_i \leq l$ are coset leaders, and let C_i have minimal trellis T_i. The subcode C_0 is chosen so that the maximum state complexity is N (occurring at some time index, say, m), where N divides M the maximum state complexity of the conventional trellis at that time index. The subcodes $C_0, C_1, \ldots C_l$ are all disjoint subcodes whose union is C. Further, the minimal trellises for $C_0, C_1, \ldots C_l$ are all structurally identical and two way proper. (That they are structurally identical can be verified by relabeling a path labeled $g_1 g_2 \ldots g_n$ in C_0 with $g_1 + h_{i_1}, g_2 + h_{i_2} \ldots g_n + h_{i_n}$ in the trellis corresponding to $C_0 + h_i$ where $h_i = h_{i_1} h_{i_2} \ldots h_{i_n}$.) We therefore refer to $T_1, T_2, \ldots T_l$ as *copies* of T_0.

Definition 4. *An* overlayed *proper trellis is said to exist for C with respect to the partition $C_0, C_1, \ldots C_l$ where $C_i, 0 \leq i \leq l$ are subcodes as defined above, corresponding to minimal trellises $T_0, T_1, \ldots T_l$ respectively,with $S_{max}(T_0) = N$, iff it is possible to construct a proper trellis T_v satisfying the following properties:*

1. *The trellis T_v has $l+1$ start states labeled $[s_0, \emptyset, \emptyset, \ldots \emptyset], [\emptyset, s_1, \emptyset \ldots \emptyset] \ldots [\emptyset, \emptyset, \ldots \emptyset, s_l]$ where s_i is the start state for subtrellis $T_i, 1 \leq i \leq l$.*
2. *The trellis T_v has $l+1$ final states labeled $[f_0, \emptyset, \emptyset, \ldots \emptyset], [\emptyset, f_1, \emptyset, \ldots \emptyset], \ldots [\emptyset, \emptyset, \ldots \emptyset, f_l]$, where f_i is the final state for subtrellis T_i, $0 \leq i \leq l$.*
3. *Each state of T_v has a label of the form $[p_0, p_1, \ldots p_l]$ where p_i is either $\emptyset$ or a state of $T_i, 0 \leq i \leq l$. Each state of T_i appears in exactly one state of T_v.*

4. *There is a transition on symbol a from state labeled* $[p_0, p_1, \ldots p_l]$ *to* $[q_0, q_1, \ldots q_l]$ *in* T_v *if and only if there is a transition from* p_i *to* q_i *on symbol a in* T_i*, provided neither* p_i *nor* q_i *is* $\emptyset$*, for at least one value of* i *in the set* $\{0, 1, 2, \ldots l\}$.
5. *The maximum width of the trellis* T_v *at an arbitrary time index* $i, 1 \leq i \leq n-1$ *is at most* N.
6. *The set of paths from* $[\emptyset, \emptyset, \ldots s_j, \ldots \emptyset]$ *to* $[\emptyset, \emptyset, \ldots, f_j, \ldots \emptyset]$ *is exactly* $C_j, 0 \leq j \leq l$.

Let the *state projection* of state $[p_0, p_1, \ldots, p_i, \ldots, p_l]$ into subcode index i be p_i if $p_i \neq \emptyset$ and empty if $p_i = \emptyset$. The *subcode projection* of T_v into subcode index i is defined by the symbol $|T_v|_i$ and consists of the subtrellis of T_v obtained by retaining all the non $\emptyset$ states in the state projection of the set of states into subcode index i and the edges between them. An overlayed trellis satisfies the property of *projection consistency* which stipulates that $|T_v|_i = T_i$. Thus every subtrellis T_j is embedded in T_v and can be obtained from it by a projection into the appropriate subcode index. We note here that the conventional trellis is equivalent to an overlayed trellis with $M/N = 1$.
Figure 3 shows the subtrellises for component codes C_0 and C_1 of the linear code defined earlier, overlayed to obtain the tailbiting trellis in Figure 4.

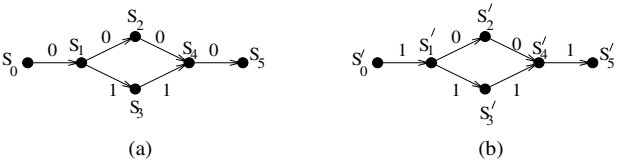


Fig. 3. Minimal trellises for (a) $C_0 = \{0000, 0110\}$ and (b) $C_1 = \{1001, 1111\}$

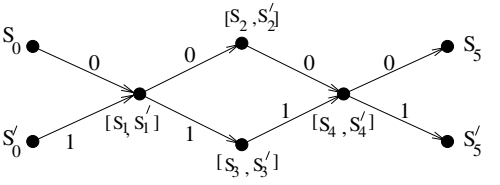


Fig. 4. Trellis obtained by overlaying trellis in figures 3(a) and 3(b)

It is shown that not all decompositions give overlayed trellises satisfying the specified bound on the width. Necessary and sufficient conditions for a decomposition to yield a tailbiting trellis with a specified bounded width are also given

in [6]. For purposes of decoding, we need the decomposition of the original conventional trellis into subtrellises that can be overlayed to form a tailbiting trellis. Each subtrellis has been shown to correspond to a coset of a group, and can be generated from the appropriate coset leader.

4 Decoding

Decoding refers to the process of forming an estimate of the transmitted codeword x from a possibly garbled version y. The received vector consists of a sequence of n real numbers where n is the length of the code. The soft decision decoding algorithm can be viewed as a shortest path algorithm on the trellis for the code. Based on the received vector, a cost $l(u, v)$ can be associated with an edge from node u to node v. The well known Viterbi decoding algorithm [20] is essentially a dynamic programming algorithm, used to compute a shortest path from the source to the goal node. Define a *winning* path as a shortest path from one of the start nodes to a final node. For a tailbiting trellis, decoding is complicated by the fact that non-accepting paths in the overlayed automaton share states with accepting paths. Thus a winning path at a goal node may not be an accepting path. We have designed and implemented a two phase algorithm which outputs a winning accepting path. The algorithm is outlined in [6] and described in detail along with proofs of correctness in [5]. We describe it informally here, along with a tiny example.

During the first phase, a Viterbi algorithm is run on the trellis and *survivors* i.e. shortest paths from any source node to all nodes are computed. Each node stores the cost of the survivor at itself at the end of the first phase. The second phase is an adaptation of the A^* algorithm[9], well known in the artificial intelligence community, and may be viewed as an adaptation of the Dijkstra algorithm with node to goal estimates added to source to node costs. All survivors at goal nodes are gathered at the end of the first phase. We term accepting paths as $s_i - f_i$ paths and non accepting paths as $s_i - f_j$ paths, with $i \neq j$. The second phase only needs to look at subtrellises T_j such that the winning path in T_j is an $s_i - f_j$ path and such that there are no $s_k - f_k$ paths with smaller cost,(we call such trellises, *residual* trellises) as the cost of a winning $s_i - f_j$ path will be an underestimate of that of the winning $s_j - f_j$ path. Any $s_i - f_j$ path with estimated cost greater than an $s_k - f_k$ path can therefore never be a winner. All residual trellises are candidates for the second phase. Decoding begins at the best candidate, i.e the one with the least estimate. The current estimates of all other residual trellises are stored in a heap. If at any instant the estimated cost of the current trellis exceeds the minimum value on the heap, the search in the current trellis is terminated, its estimate inserted into the heap, and the trellis corresponding to the minimum value in the heap taken up for searching next. Thus the algorithm makes its way towards the goal travelling on the best subtrellis seen so far at any given instant. As soon as the goal node is reached, we are sure that we have the

winning path. For high signal to noise ratios it is observed that the algorithm either does not need the second phase at all, or that it usually stays on a single subtrellis for the whole of the second phase. We illustrate with a tiny example. Though this one has only one residual trellis, it serves to illustrate the idea.

Figure 5 gives an overlayed trellis with some hypothetical costs. The nodes are labeled with survivors(within parentheses) after the first phase. Since the four codewords have costs 11,9,10,12, the winning path is $acefg$ with a cost of 9. The first phase outputs winners with cost 6 at g and 10 at h, corresponding to paths $bcefg$ and $bcefh$. Subtrellis corresponding to $s_i - f_j$ pair $b-g$ is a residual trellis so we begin decoding at a with estimate 6. At c the estimate changes to 4+(6-1)=9; at d it is 8+(6-5)=9; at e it is 5-(6-2)=9; at f it is min(9+(6-4), 7+(6-4))=9; at g it is 9+0=9. Hence the winning path is $acefg$.

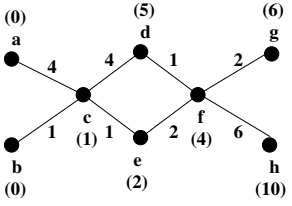


Fig. 5. Tailbiting trellis with hypothetical edge costs and survivors after first phase

5 Implementation

The package has been implemented in C.The minimal trellis construction algorithm of Kschischang-Sorokine that is implemented has been validated by generating minimal trellises for several codes for which these structures have been published, among them the (48,24) quadratic residue code, the (24,12) Golay code and the (16,7) lexicode(for which the state complexity profiles are available in [19].)

For the decomposition, subtrellises are generated from a tailbiting trellis by carrying out a forward traversal from each start state and keeping track of which nodes reachable from the start state also reach the appropriate final state. Thus the complexity of subtrellis generation from a tailbiting trellis is $O(s)$ where s is the number of states. Additional storage is not required for the subtrellises as each node of the tailbiting trellis has a vector of length $l+1$ associated with it (where $l+1$ is the number of subtrellises), which indicates whether a node of the tailbiting trellis is present in a certain subtrellis or not.

For the decoding, the channel is modeled as an AWGN channel using a random number generator that produces numbers that are normally distributed with mean 0 and variance $\frac{N_0}{2}$, where N_0 is the noise energy level. The tail-biting trellis is implemented as a two dimensional array of states reprepresenting vertices. Each state contains incoming and outgoing branches. Branches contain labels that correspond to a codeword symbols. If there are $l+1$ subtrellises, each state has storage for the $l+2$ surivivor paths (one obtained in the first phase and the remaining $l+1$ for each individual subtrellis to be used in the second phase), and for the corresponding metric. The metric along a branch is computed at runtime depending on the generated random codeword. Each state contain a membership array of length $l+1$ to indicate the membership in subtrellises. If vertex v belongs to subtrellis i, then the i'th bit of the array is set to 1, otherwise it is set to 0. The heap required in the second phase is implemented as an array.

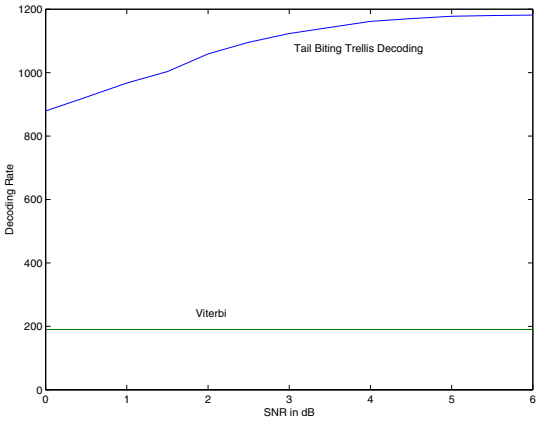


Fig. 6. Rates of decoding using the Viterbi algorithm and the two phase algorithm for the Golay code

5.1 Simulation Results

We present the simulation results of the algorithm tested on the extended (24,12) Golay Code. The rate of the code is $\frac{1}{2}$. The minimal tail-biting trellis of the 12-section (24,12) Golay code has a uniform state complexity of 16 at all time indices and is described in [3]. The conventional 12-section trellis of the (24,12) Golay Code has the state complexity profile (1,4,16,64,256,256,256,256,256,64,16,4,1). The conventional trellis has 1066 states and 2728 branches and the tail-biting trellis has 208 states and 384 branches. Each branch of the 12-sectioned Golay code represents 2 code bits.

A large number of random codewords is generated for various signal-to-noise ratios and the average rate of decoding is calculated by running the algorithm on

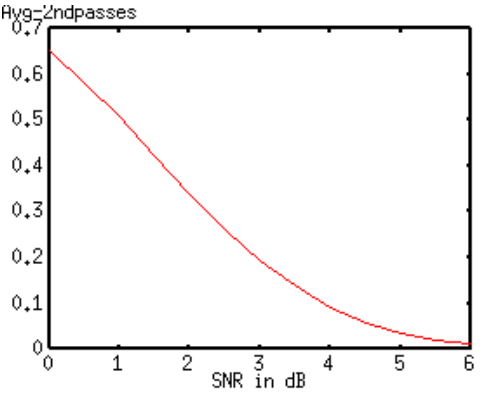


Fig. 7. Probability of a second pass in the two phase algorithm as a function of SNR

the tail-biting trellis. The results are compared with the Viterbi rate of decoding on the conventional trellis. The result is presented in figure 6. The rate graph shows that the two phase algorithm on the tailbiting trellis is significantly better than Viterbi decoding on the conventional trellis even at a low SNR of 0dB. While the Viterbi rate of decoding remains constant around 190 codewords/sec for SNR values in the range [0,6], the rate of the proposed algorithm increases steadily from 879 to 1181 codewords/sec. It is known[19] that the Viterbi algorithm on a trellis with V nodes and E edges requires $|E|$ multiplications and $|E| - |V| + 1$ additions. From the vertex and edge counts for the conventional and tailbiting trellises for the Golay code given above, we conclude that the overheads in heap operations and the number of switches are not significant.

It is also seen that for high SNR values, the decoding rarely needs a second phase. From figure 7 we see that the probability that the algorithm requires a second phase decreases from 0.652 to 0.012 as SNR increases from 0 to 6dB. During the second phase the algorithm switches from one subtrellis to other if there is a subtrellis on top of the heap with smaller metric than the subtrellis that is currently being expanded. Figure 8 shows that the average number of switches in the second pass decreases steadily to 1.53 showing that the search in second phase is usually restricted to a single subtrellis for large SNR values.

The measure of the probabilty of decoding error of a maximum likelihood decoder is given by the Bit Error Rate (BER). For a codeword C of an (n, k) code with each codeword symbol requiring m bits, mn bits are transmitted. If the decoder decodes to codeword C', whose mn bits differ from C in e locations, the bit error rate is $\frac{e}{mn}$. Thus, BER is the decoding error per bit. The bit error rate for the tail-biting trellis decoding is presented in figure 9. The probability of decoding error is very small for large SNR values, and the curve is consistent with others obtained in the literature for the Golay code .

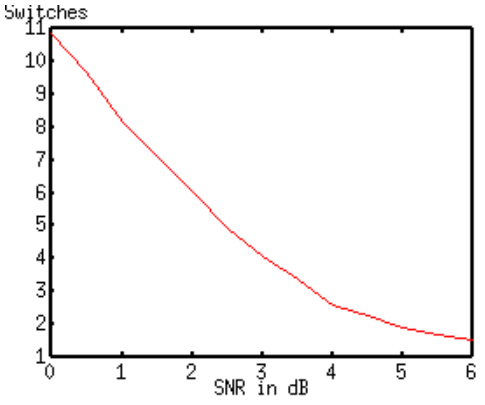


Fig. 8. Average number of switches between trellises when there is a second pass

6 Conclusion

We have implemented a package for the implementation of block codes as trellises and an efficient decoding algorithm and simulator for tailbiting trellises. Inclusion of a module for the conversion from conventional to tailbiting trellises when the full theory is available will make this, we hope, an useful general purpose tool for the coding community.

References

1. L.R.Bahl, J.Cocke, F.Jelinek, and J. Raviv, Optimal decoding of linear codes for minimizing symbol error rate, *IEEE Trans. Inform. Theory* **20**(2), March 1974, pp 284-287.
2. R.E. Blahut, *Theory and Practice of Error control Codes*, Addison Wesley, 1984.
3. A.R.Calderbank, G.David Forney,Jr., and Alexander Vardy, Minimal Tail-Biting Trellises: The Golay Code and More, *IEEE Trans. Inform. Theory* **45**(5) July 1999,pp 1435-1455.
4. G.C.Clark, Jr. and J.B. Cain, *Error-Correction Coding for Digital Communication.*, New York: Plenum, 1981.
5. Amitava Dasgupta, Priti Shankar, Kaustubh Deshmukh, and B.S,Rajan *On Viewing Block Codes as Finite Automata,* Technical Report IISc-CSA-99-7, Department of Computer Science and Automation, Indian Institute of Science, Bangalore-560012,
6. K.Deshmukh, Shankar,P., Dasgupta,A.,Sundar Rajan,B., On the many faces of block codes, in *Proceedings of STACS 2000, LNCS 1770* , (Lille, France, February 2000), pp 53-64.
7. G.D. Forney, Jr.,Coset codes II: Binary lattices and related codes, *IEEE Trans. Inform. Theory* **36**(5), Sept. 1988,pp 1152-1187.

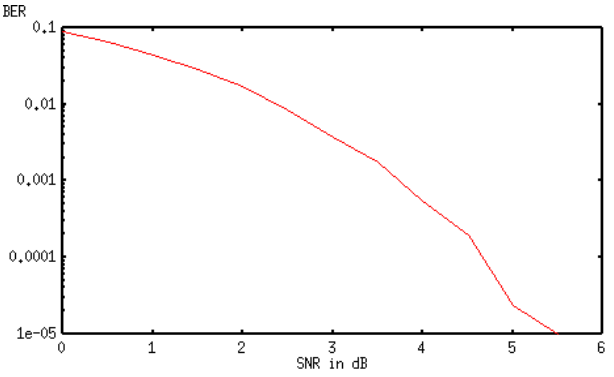


Fig. 9. Variation of the bit error rate with SNR

8. G.D. Forney, Jr. and M.D. Trott, The dynamics of group codes:State spaces, trellis diagrams and canonical encoders, *IEEE Trans. Inform. Theory* **39**(5) Sept 1993,pp 1491-1513.
9. P.E. Hart, N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Solid-State Circuits* **SSC-4**, 1968, pp 100-107.
10. Ralf Kotter and Vardy, A.,Construction of Minimal Tail-Biting Trellises,in *Proceedings IEEE Information Theory Workshop* (Killarney, Ireland, June 1998), 72-74.
11. Ralf Kotter and Vardy, A., The theory of tailbiting trellises, (manuscript in preparation).
12. F.R.Kschischang and V.Sorokine, On the trellis structure of block codes, *IEEE Trans. Inform. Theory* **41**(6), Nov 1995,pp 1924-1937.
13. D.Lind and M.Marcus, *An Introduction to Symbolic Dynamics and Coding*, Cambridge University Press, 1995.
14. F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error Correcting Codes*, North-Holland, Amsterdam, 1981.
15. J.L.Massey, Foundations and methods of channel encoding, in *Proc. Int. Conf. on Information Theory and Systems* **65**(Berlin, Germany) Sept 1978.
16. R.J.McEliece, *The Theory of Information and Coding*, Encyclopedia of Mathematics and its Applications, Addison Wesley, 1977.
17. R.J.McEliece, On the BCJR trellis for linear block codes, *IEEE Trans. Inform. Theory* **42**,November 1996, pp 1072-1092.
18. D.J. Muder, Minimal trellises for block codes, *IEEE Trans. Inform. Theory* **34**(5), Sept 1988,pp 1049-1053.
19. A.Vardy, Trellis structure of codes, in *Handbook of Coding Theory*,V.S. Pless and W.C. Huffman, Eds.,Elsevier Science, 1998.
20. A.J. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, *IEEE Trans. Inform. Theory* **13**, April 1967, pp 260-269.
21. N.Wiberg, H.-A. Loeliger and R.Kotter, Codes and iterative decoding on general graphs, *Eoro. Trans. Telecommun.*,**6** pp513-526, Sept 1995.
22. J.K. Wolf, Efficient maximum-likelihood decoding of linear block codes using a trellis, *IEEE Trans. Inform. Theory* **24** pp 76-80.

Regional Least-Cost Error Repair

M. Vilares, V.M. Darriba, and F.J. Ribadas

Computer Science Department, Campus de Elviña s/n, 15071 A Coruña, Spain
vilares@udc.es, {darriba,ribadas}@dc.fi.udc.es

Abstract. We describe an algorithm to deal with automatic error repair over unrestricted context-free languages. The method relies on a regional least-cost repair strategy with validation, gathering all relevant information in the context of the error location. The system guarantees the asymptotic equivalence with global repair strategies.

1 Introduction

Until recently, errors were simply recovered by the consideration of fiducial symbols to provide mile-posts for error recovery, which allows a reduction in time and space bounds, although it is not always easy to determine if all relevant information to the error recovery process has been seen [3]. The significant reduction in cost processing has propitiated a renewable interest in methods that take into account the constraints on context. We can differentiate [3] two families of algorithms: one class, called *local repair*, make modifications to the input so that at least one more original input symbol can be accepted by the parser. The simplicity of these methods sometimes causes them to choose a poor repair [4, 1].

In contrast to local techniques, the *global repair* algorithms examine the entire program and make a minimum of changes to repair all the syntax errors, although they expend equal effort on all parts of the program, including areas that contain no errors. In between the local and global methods, Levi [2], suggested *regional repair* algorithms that fix a portion of the program including the error and as many additional symbols as needed to assure a good repair. In relation to global and local methods, the regional algorithms must answer the additional question of determining just how large a region to repair.

In addition, when several repairs are available, the system must provide some method of choosing among them, and a common strategy is to assign individual costs. A repair algorithm that guarantees finding the lowest-cost repair possible, is called a *least-cost* repair algorithm. Our proposal is a regional least-cost strategy which applies a dynamic validation in order to avoid cascaded errors.

2 A Dynamic Frame for Parsing

We introduce our parsing frame, as implemented in Ice [5]. Our aim is to parse sentences in the language $\mathcal{L}(\mathcal{G})$ generated by a context-free grammar

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 293–301, 2001.

$\mathcal{G} = (N, \Sigma, P, S)$, where N is the set of non-terminals, Σ the set of terminal symbols, P the rules and S the start symbol. The empty string will be represented by ε.

2.1 The Operational Model

We assume that we produce a *push-down automaton* (PDA) from $\mathcal{G}$. In practice, we chose an LALR(1) device, which is possibly non-deterministic, for the language $\mathcal{L}(\mathcal{G})$. Formally, a PDA is a 7-tuple $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, \delta, q_0, Z_0, \mathcal{Q}_f)$ where: $\mathcal{Q}$ is the set of states, Σ the set of input symbols, Δ the set of stack symbols, q_0 the initial state, Z_0 the initial stack symbol, $\mathcal{Q}_f$ the set of final states, and δ a finite set of transitions of the form $p\ X\ a \mapsto q\ Y$ with $p, q \in \mathcal{Q}$, $a \in \Sigma \cup \{\varepsilon\}$ 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, ax is the remaining input where the symbol a is the next to be shifted, $x \in \Sigma^*$. The application of $p\ X\ a \mapsto q\ Y$ results in a new configuration $(q, Y\alpha, x)$ where the terminal symbol a has been scanned, X has been popped, and Y has been pushed.

The algorithm proceeds by building a collection of *items*, compact representations of the recognizer stacks, by applying transitions to existing ones, until no new application is possible. The algorithm associates a set of items S_i^w, called *itemset*, for each input symbol w_i at the position i in the input string of length n, $w_{1..n}$. An item is of the form $[p, X, S_j^w, S_i^w]$, where $p \in \mathcal{Q}$, $X \in \Delta$, S_j^w is the *back pointer* to the itemset associated to the symbol w_i at which we began to look for that configuration of the automaton, and S_i^w is the current itemset.

2.2 The Recognizer

Formally, given a transition $\tau = \delta(p, X, a) \ni (q, Y)$, we translate it to items of the following form:

$$
\begin{array}{lll}
1.\ \tilde{\delta}([p, X, S_j^w, S_i^w], a) \ni [q, \varepsilon, S_i^w, S_i^w], & & \textbf{if } Y = X \\
2.\ \tilde{\delta}([p, X, S_j^w, S_i^w], a) \ni [p, Y, S_i^w, S_{i+1}^w], & & \textbf{if } Y = a \\
3.\ \tilde{\delta}([p, X, S_j^w, S_i^w], a) \ni [p, Y, S_i^w, S_i^w], & & \textbf{if } Y \in N \\
4.\ \tilde{\delta}([p, \varepsilon, S_j^w, S_i^w], a) & \ni \tilde{\delta}_d([q, \varepsilon, S_l^w, S_j^w], a) \ni [q, \varepsilon, S_l^w, S_i^w], & \textbf{if } Y = \varepsilon \\
& \forall q \in \mathcal{Q} \text{ such that } \exists\, \delta(q, X, \varepsilon) \ni (p, X) &
\end{array}
$$

with $\tilde{\delta} : \mathrm{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \{\mathrm{It} \cup \tilde{\delta}_d\}$ and $\tilde{\delta}_d : \mathrm{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \mathrm{It}$, where *It* is the set of all items developed in the parsing process and $\tilde{\delta}_d$ is called the set of *dynamic transitions*. Succinctly, we can describe the preceding cases as follows:

1. A goto action from the state p to state q under transition X.
2. A push of a from state p. The new item belongs to itemset S_{i+1}^w.
3. A push of non-terminal Y from state p.
4. A pop action from state p, where q is an ancestor of state p under transition X. We generate a *dynamic transition* $\tilde{\tau}_d$ to treat the absence of information about the rest of the stack. This transition is applicable not only to the

configuration resulting from the first one, but also on those to be generated and sharing the same syntactic structure.

Fairness and completeness of the dynamic construction are guaranteed by an equitable selection order *It*. To ignore redundant items we use a subsumption relation based on equality. Authors prove in [5] that time and space bounds are, in the worst case, $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ respectivley, for inputs $w_{1..n}$.

3 Regional Least-Cost Error Repair

Following Mauney and Fischer in [3], 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. So, we can talk about the *point of error* as the point at which the difference occurs. The *point of detection* is the point at which the parser detects that there is an error in the input and calls the repair algorithm.

Definition 1. *Let $w_{1..n}$ be an input string, we say that w_i is a* point of error *iff:* $\nexists\, [p, \varepsilon, S_l^w, S_i^w] \in S_i^w / \delta(p, X, w_i) = (q, w_i)$

The point of error is easily fixed by the parser itself and, in order to locate the origin of the error at minimal cost, we should try to limit the impact on the parse, focusing on the context of subtrees close to the point of error.

Definition 2. *Let w_i be a point of error for the input string $w_{1..n}$, we define the set of* points of detection associated to w_i*, as follows:*

$$detection(w_i) = \{w_{i'} / \exists A \in N,\ A \overset{+}{\Rightarrow} w_{i'} \alpha w_i\}$$

and we say that $A \overset{+}{\Rightarrow} w_{i'} \alpha w_i$ is a derivation defining the point of detection $w_{i'} \in detection(w_i)$.

Intuitively, the error is located in the immediate left parse context, represented by the closest viable node, or in the immediate right context, represented by the lookahead. However, sometimes can be usefull to isolate the parse branch in which the error appears.

Definition 3. *Let w_i be a point of error for $w_{1..n}$, we say that $[p, X, S_l^w, S_i^w] \in S_i^w$ is an* error item *iff:* $\exists\ a \in \Sigma,\ \delta(p, \varepsilon, a) \neq \emptyset$*, and we say that $[p, \varepsilon, S_{i'}^w, S_{i'}^w] \in S_{i'}^w$ is a* detection item *associated to w_i iff* $\exists\ a \in \Sigma, \delta(p, A, a) \neq \emptyset,\ A \in N$ *defining w_i, such that:*

$$\begin{array}{ll}
\delta(q_1, \varepsilon, w_{i'}) \ni (q_1, B_2), & \delta(q_1, B_2, w_{i'}) \ni (q_2, \varepsilon) \\
\vdots & \vdots \\
\delta(q_{n-1}, \varepsilon, w_{i'}) \ni (q_{n-1}, B_n), & \delta(q_{n-1}, B_n, w_{i'}) \ni (q_n, \varepsilon) \\
\delta(q_n, \varepsilon, w_{i'}) \ni (q_n, w_{i'}), & B_i \overset{+}{\Rightarrow} \varepsilon,\ \forall i \in [1, n]
\end{array}$$

We talk about *error and detection items*, when they represent nodes including the recognition of points of error and detection, respectively. The condition for error items implies that no scan action is possible for token w_i. In the detection case, conditions look for items recognizing a point of detection on a parse branch including an error item in w_i, disregarding empty reductions which are not relevant for this purpose.

Definition 4. *A* modification M to a string *of length* n, $w_{1..n} = w_1 \dots w_n$, *is a series of edit operations,* $E_1 \dots E_n E_{n+1}$, *in which each* E_i *is applied to* w_i *and possibly consists of a series of insertions before* w_i, *replacements or deletion of* w_i. *The string resulting from the application of the modification* M *to the string* w *is written* $M(w)$.

We now restrict the notion of modification to focus on a given zone of the input string, introducing the concept of error repair in this space. Intuitively, we look for conditions that guarantee the ability to recover the parse from the error, at the same time as it allows us to isolate repair branches by using the concept of reduction. We are also interested in minimizing the structural impact in the parse tree, and finally in introducing the notion of scope as the lowest reduction summarizing the process at a point of detection.

Definition 5. *Let* x *be a valid prefix in* $\mathcal{L}(\mathcal{G})$, *and* $w \in \Sigma^*$, *such that* xw *is not a valid prefix in* $\mathcal{L}(\mathcal{G})$. *We define a* repair of w following x *as* $M(w)$, *so that:*

$$\exists A \in N \Big/ \begin{array}{l} S \stackrel{+}{\Rightarrow} x_{1..i-1}A \stackrel{+}{\Rightarrow} x_{1..i-1}x_{i..m}M(w),\ i \leq m \\ B \stackrel{*}{\Rightarrow} \alpha A \beta,\ \forall B \stackrel{+}{\Rightarrow} x_{j..m}M(w),\ j < i \\ A \stackrel{*}{\Rightarrow} \gamma C \rho,\ \forall C \stackrel{+}{\Rightarrow} x_{i..m}M(w) \end{array}$$

We denote the set of repairs of w *following* x *by* $\mathrm{repair}(x, w)$, *and* A *by* $\mathrm{scope}(M)$.

However, the notion of *repair*(x, w) is not sufficient for our purposes, since our aim is to extend the error repair process to consider all possible points of detection proposed by the algorithm for a given point of error, which implies simultaneously considering different valid prefixes and repair zones.

Definition 6. *Let* $e \in \Sigma$ *be a point of error, we define the set of* repairs for e, *as repair*$(e) = \{xM(w) \in \textit{repair}(x, w)/w_1 \in \textit{detection}(e)\}$, *where* detection$(e)$ *denotes the set of points of detection associated to* e.

We now need a mechanism to filter out undesirable repair processes, in order to reduce the computational charges. To do that, we should introduce comparison criteria to only select those repairs with minimal cost.

Definition 7. *For each* $a \in \Sigma$ *we assume the existence of positive insert,* $I(a)$; *delete,* $D(a)$, *and replace* $R(a)$ *costs*[1]. *The* cost of a modification[2] $M(w_{1..n})$ *is*

[1] if any edit operation is not applied, we assume its cost to be zero.

[2] we assume that delete and replace operations are exclusive for the same token.

given by $\text{cost}(M(w_{1..n})) = \Sigma_{i=1}^{n}(\Sigma_{j\in J} I(w_i^j) + D(w_i) + R(w_i))$. *In particular,* $\Sigma_{j\in J} I(w_i^j)$ *means that several insertion hypotheses are possible before the token* w_i *is real.*

When several repairs are available on different points of detection, we need a condition to ensure that only those with the same minimal cost are considered, looking for the best repair quality.

Definition 8. *Let* $e \in \Sigma$ *be a point of error, we define the set of* regional repairs for e*, as follows:*

$$regional(e) = \{xM(w) \in repair(e) \Big/ \begin{matrix} cost(M) \leq cost(M'),\ \forall M' \in repair(x,w) \\ cost(M) = \min_{L\in repair(e)}\{cost(L)\} \end{matrix} \}$$

It is also necessary to take into account the possibility of cascaded errors, that is, errors precipitated by a previous erroneous repair diagnostics. Previous to dealing with the problem, we need to establish the existing relationship between the regional repairs for a given point of error, and future points of error.

Definition 9. *Let* w_i, w_j *be points of error in an input string* $w_{1..n}$*, such that* $j > i$*. We define the set of* viable repairs for w_i in w_j*, as follows:*

$$viable(w_i, w_j) = \{xM(y) \in regional(w_i)/xM(y)\ldots w_j \text{ valid prefix for } \mathcal{L}(\mathcal{G})\}$$

Intuitively, the repairs in $viable(w_i, w_j)$ are the only ones capable of ensuring the continuity of the parse in $w_{i..j}$ and, therefore, the only possible repairs at the origin of the phenomenon of cascaded errors.

Definition 10. *Let* w_i *be an point of error for the input string* $w_{1..n}$*, we say that a point of error* w_j*,* $j > i$ *is a* point of error precipitated by w_i *iff*

$$\forall xM(y) \in viable(w_i, w_j),\ \exists A \in N \text{ defining } w_{j'} \in detection(w_j)$$

such that $A \overset{+}{\Rightarrow} \beta scope(M)\ldots w_j$.

Intuitively, a point of error w_j is precipitated by the result of previous repairs on a point of error w_i, when all reductions defining points of detection for w_j summarize some viable repair for w_i in w_j.

4 The Algorithm

We propose that the repair be obtained by searching the PDA itself to find a suitable configuration to allow the parse to continue. At this point, our approach agrees with McKenzie *et al.* in [4], although this method is not asymptotically equivalent to a global repair strategy, and introduces an unsafe technique to speed up the repair algorithm [1]. In relation to this last, McKenzie *et al.* propose a pruning mechanism in order to reduce the number of configurations to be dealt

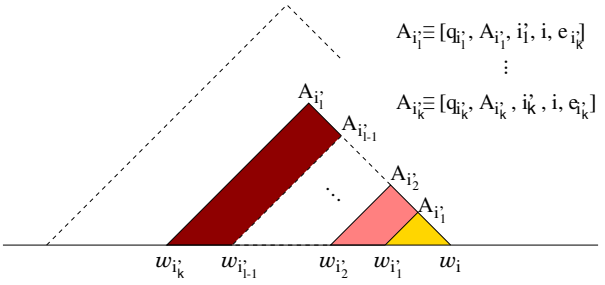


Fig. 1. Error detection

with during the repair process. This mechanism may lead to suboptimal regional repairs or may cause failure to produce any repair even if an error exists.

The problem due to pruning is based on a simple condition that ignores a stack configuration if an earlier one had the same stack top. The motivation is that this newer configuration would not lead to any cheaper repairs than the older one. Our dynamic programming construction eliminates this problem by considering all possible repair paths.

4.1 A Simple Case

We assume that we deal with the first error detected in the input string. The major features of the algorithm involve beginning with a list of error items, with an error counter zero. In order to compute the error counter, we extend the item structure: $[p, X, S_i^w, S_j^w, \mathrm{e}]$, where now e is the error counter accumulated in the recognition of $X \in N \cup \Sigma$.

For each error item, we successively investigate the corresponding list of detection items, one for each parse branch including the error item. One a point of error w_i has been fixed, we can associate to it different points of detection $w_{i'_1}, \ldots w_{i'_k}$, as is shown in Fig. 1. Detection items are located by using the back pointer, that indicates the itemset where we have applied the last PDA action. So, we recursively go back into its ancestors until we find the first descendant of the last node that would have to be reduced if the lookahead was correct[3].

Once the detection items have been fixed for the corresponding error item, on each of the parse branches relying on them we apply all possible transitions beginning at the point of detection. These transitions correspond to four error hypotheses, from a given item:

– For scan transitions the item obtained is the same as for standard parsing.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{scan}\, w_i}{\longrightarrow} [p, w_i, S_i^w, S_{i+1}^w, 0]$$

[3] this information is directly obtained from the PDA.

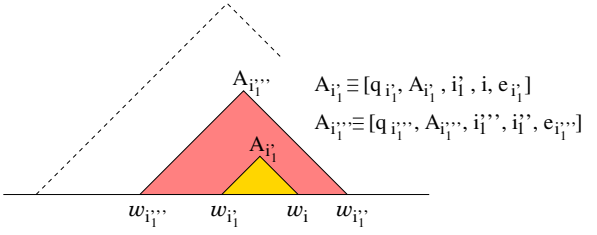


Fig. 2. Repair scope

– In the case of insertion hypothesis, we initialize the error counter by taking into account the cost of the inserted token, which is included as stack symbol, and we add the new item to the same itemset, preserving the back pointer.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{insert } a}{\longrightarrow} [p, a, S_i^w, S_i^w, I(a)],\ \delta(p, \varepsilon, a) \neq \emptyset$$

– For deletion hypothesis, we initialize the error counter by taking into account the cost of the deleted token and we add the new item to the next itemset, using the same stack symbol. The back pointer is initialized to the current itemset.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{delete } w_i}{\longrightarrow} [p, \varepsilon, S_i^w, S_{i+1}^w, D(w_i)]$$

– Finally, for mutation hypothesis, we initialize the error counter by taking into account the cost of the replaced token and we add the new item to the next itemset. The back pointer is initialized to the current itemset, and the new token resulting from the mutation is included as stack symbol.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{replace } w_i \text{ by } a}{\longrightarrow} [p, a, S_i^w, S_{i+1}^w, R(a)],\ \delta(p, \varepsilon, a) \neq \emptyset$$

We do that until a reduction verifying definition 5 covers both error and detection items accepting a token in the remaining input string, as is shown in Fig. 2, where $[w_{i'''_1}, w_{i''_1}]$ delimits the scope of a repair detected at the point $w_{i'_1} \in \mathit{detection}(w_i)$. Once we have applied the previous methodology to each detection item considered, we take only those repairs with regional lowest cost, applying definition 8. At this moment the parse goes back to standard mode.

We use a bottom-up strategy not only to parse the input, but also to compute error counters. This establishes a difference with McKenzie *et al.* [4], that uses a bottom-up parsing architecture with a top-down computation of the error counters. An inheritance strategy to compute the error counters, make the task of propagating these counters on shared parse branches a complex one. In our case, error counters are initialized at each error hypothesis and summarized only at reduce actions time. So, dynamic transitions must include information about the accumulated error counter in the part of the reduce action to be shared. The process is illustrated in Fig. 3 for two reductions, A_i and A_j, over a same rule $A \to X_1 \ldots X_m$ sharing the last $X_{k+1} \ldots X_m$ syntactic categories. We re-take the part of the error counter accumulated during the first reduction, $e_{i'_{k+1}} + \ldots + e_{i'_m}$, for these common categories.

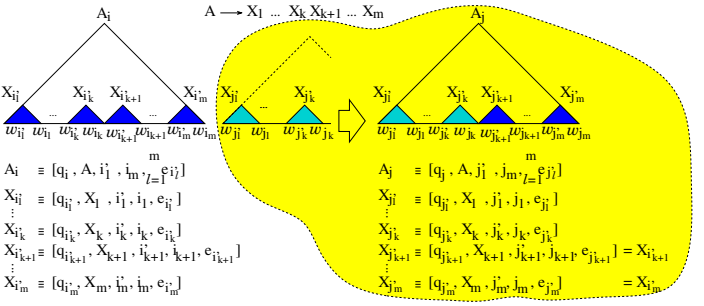


Fig. 3. Dynamic transitions in repair mode

4.2 The General Case

We now assume that the current repair process is not the first one and, therefore, can modify a previously repaired string. This arises when we realize that we come back to a detection item for which any parse branch includes a previous repair process. This process is illustrated in Fig. 4 for a point of error w_j precipitated by w_i, showing how the variable $A_{j'_1}$ defining w_j summarizes $A_{i'''_1}$, the scope of a previous repair defined by $A_{i'_1}$.

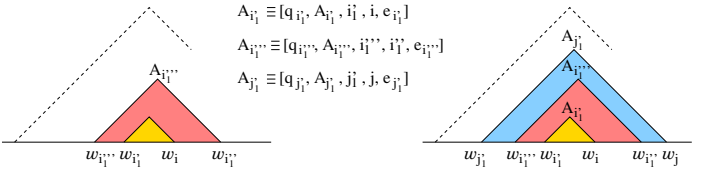


Fig. 4. Dealing with precipitated errors

To deal with precipitated errors, the algorithm re-takes the previous error counters, adding the cost of the new error repair hypothesis to profit from the experience gained from previous repair processes. At this point, regional repairs have two important properties. First, it is independent of the shift-reduce parsing algorithm used. The second property is a consequence of the lemma below.

Lemma 1. *(The Expansion Lemma) Let w_i, w_j be points of error in $w_{1..n} \in \Sigma^*$, such that w_j is precipitated by w_i, then*

$$min\{j'/w_{j'} \in detection(w_j)\} < min\{i'/w_{i'} = y_1,\ xM(y) \in viable(w_i, w_j)\}$$

Proof. Let $w_{i'} \in \Sigma$, such that $w_{i'} = y_1,\ xM(y) \in viable(w_i, w_j)$ be a point of detection for w_i, for which some parsing branch derived from a repair in $regional(w_i)$ has successfully arrived at w_j.

Let w_j be a point of error precipitated by $xM(y) \in \mathit{viable}(w_i, w_j)$. By definition, we can assure that

$$\exists B \in N/B \stackrel{+}{\Rightarrow} w_{j'}\alpha w_j \stackrel{+}{\Rightarrow} \beta \mathrm{scope}(M) \ldots w_j \stackrel{+}{\Rightarrow} \beta x_{l..m} M(y) \ldots w_j,\ w_{i'} = y_1$$

Given that $\mathit{scope}(M)$ is the lowest variable summarizing $w_{i'}$, it immediately follows that $j' < i'$, and we conclude the proof by extending the proof to all repairs in $\mathit{viable}(w_i, w_j)$. □

Corollary 1. *Let w_i, w_j be points of error in $w_{1..n} \in \Sigma^*$, such that w_j is precipitated by w_i, then*

$$\mathit{max}\{\mathrm{scope}(M),\ M \in \mathit{viable}(w_i, w_j)\} \subset \mathit{max}\{\mathrm{scope}(\tilde{M}),\ \tilde{M} \in \mathit{regional}(w_j)\}$$

Proof. It immediately follows from lemma 1. □

This allow us to get an asymptotic behavior close to global repair methods. This property has profound implications for the efficiency, measured by time and space taken, the simplicity and the power of computing regional repairs.

Corollary 2. *Let $w_{1..n}$ be an input string with a point of error in w_i, $i \in [1, n]$, then the time and space bounds for the regional repair algorithm are $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$, in the worst case, respectively.*

Proof. It immediately follows from the previous corollary 1. □

5 Conclusions

To improve the quality of repairs we should gather information to the right and to the left of the point of detection as long as this information could possibly be relevant. A criterion that meets our requirements is to expand the repair mode until it is guaranteed to accept the next input symbol, but maintains the chance of reconsidering the process once the system has detected that an incorrect repair assumption has been made.

References

1. Eberhard Bertsch and Mark-Jan Nederhof. On failure of the pruning technique in "Error Repair in Shift-Reduce Parsers". *ACM Transactions on Programming Languages and Systems*, 21(1):1–10, January 1999.
2. J.P. Levy. Automatic correction of syntax-errors in programming languages. *Acta Informatica*, 4:271–292, 1975.
3. J. Mauney and C.N. Fischer. Determining the extend of lookahead in syntactic error repair. *ACM Transactions on Programming Languages and Systems*, 10(3):456–469, 1988.
4. Bruce J. McKenzie, Corey Yeatman, and Lorraine De Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, July 1995.
5. M. Vilares and B.A. Dion. Efficient incremental parsing for context-free languages. In *Proc. of the 5^{th} IEEE International Conference on Computer Languages*, pages 241–252, Toulouse, France, 1994.

The Parameterized Complexity of Intersection and Composition Operations on Sets of Finite-State Automata

H. Todd Wareham

Department of Computer Science, Memorial University of Newfoundland,
St. John's, NF, Canada A1B 3X5
harold@cs.mun.ca

Abstract. This paper uses parameterized complexity analysis to delimit possible non-polynomial time algorithmic behaviors for the finite-state acceptor intersection and finite-state transducer intersection and composition problems. One important result derived as part of these analyses is the first proof of the NP-hardness of the finite-state transducer composition problem for both general and p-subsequential transducers.

1 Introduction

Certain applications of finite-state automata are most naturally stated in terms of the intersection or composition of a set of automata [7,10]. One approach to solving these problems is to use state Cartesian product constructions [6, pp. 59-60] to build the automaton associated with the intersection or composition and then answer the query relative to a determinized and/or minimized version of that automaton. Though such queries as emptiness or membership can typically be answered in time and space linear in the size of the derived automaton, the automaton may have $O(|Q|^{|A|})$ states, where $|A|$ is the number of automata in the set and $|Q|$ is the maximum number of states in any automaton in the set. This is to be expected, as many problems on sets of automata are NP-hard and hence do not have polynomial-time algorithms unless $P = NP$. However, are there other non-polynomial time algorithmic options for solving such problems, *e.g.*, an algorithm whose non-polynomial time complexity term is purely a function of $|Q|$ and $|\Sigma|$, where $|\Sigma|$ is the size of the language-alphabet? Knowledge of such options would be useful in practice for choosing the most efficient algorithm in situations in which one or more of the characteristics of the problem are of bounded value, *e.g.*, $|Q| \leq 4$ and $|\Sigma| \leq 26$.

In this paper, techniques from the theory of parameterized computational complexity [4] are used to determine part of the range of possible non-polynomial time algorithmic behaviors for the finite-state acceptor intersection and finite-state transducer intersection and composition problems. These analyses generalize and simplify results given in [13]. One important result derived as part of these analyses is the first proof of the NP-hardness of the finite-state transducer composition problem for both general and p-subsequential transducers.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 302–310, 2001.

1.1 Terminology

A **finite state acceptor (FSA)** is a 5-tuple $\langle Q, \Sigma, \delta, s, F\rangle$ where Q is a set of states, Σ is an alphabet, $\delta : Q \times \{\Sigma \cup \{\epsilon\}\} \times Q$ is a transition relation, $s \in Q$ is the start state, and $F \subseteq Q$ is a set of final states. If δ has no entries of the form (q, ϵ, q') and is also a function, *i.e.*, for each $q \in Q$ and $s \in \Sigma$ there is at most one state $q' \in Q$ such that $(q, s, q') \in \delta$, the FSA is a **deterministic finite-state acceptor (DFA)**.

A **finite state transducer (FST)** is a 6-tuple $\langle Q, \Sigma_i, \Sigma_o, \delta, s, F\rangle$ where Q is a set of states, Σ_i and Σ_o are the input and output alphabets, respectively, $\delta : Q \times \Sigma_i^* \times \Sigma_o^* \times Q$ is a transition relation, $s \in Q$ is the start state, and $F \subseteq Q$ is a set of final states. There are several possible definitions of determinism for FST; types of interest here are:

- ***i*-Deterministic FST (sequential FST [11])**: For each $q \in Q$ and $x \in \Sigma_i^*$, there is at most one $y \in \Sigma_o^*$ and $q' \in Q$ such that $(q, x, y, q') \in \delta$.
- ***i*/*o*-Deterministic FST**: For each $q \in Q$, $x \in \Sigma_i^*$ and $y \in \Sigma_o^*$, there is at most one $q' \in Q$ such that $(q, x, y, q') \in \delta$.

All FST in this paper are restricted to singleton labels that are ϵ-free, *i.e.*, $\delta : Q \times \Sigma_i \times \Sigma_o \times Q$. Note that such FST will always produce output strings of the same length as the input string, [7, Lemma 3.3].

2 Parameterized Complexity Analysis

The theory of NP-completeness [5] proposes a class NP of decision problems that is conjectured to properly include the class P of decision problems that have polynomial-time algorithms. For a given decision problem Π, if every problem in NP reduces[1] to Π, *i.e.*, Π is NP**-hard**, then Π does not have a polynomial-time algorithm unless $P = NP$.

It may still be possible to solve NP-hard problems by invoking non-polynomial time algorithms that are effectively polynomial time because their non-polynomial terms are purely functions of sets of aspects of the problems that are of bounded size or value in instances of those problems encountered in practice, where an **aspect** of a problem is some (usually numerical) characteristic that can be derived from instances of that problem, *i.e.*, $|Q|$, $|\Sigma|$, and $|A|$ in the case of finite-state automaton intersection and composition problems. The theory of parameterized computational complexity [4] provides explicit mechanisms for analyzing the effects of both individual aspects and sets of aspects on problem complexity.

[1] Given decision problems Π and Π', Π **reduces to** Π', *i.e.*, $\Pi \leq_m \Pi'$, if there is an algorithm A that transforms an instance x of Π into an instance y of Π' such that A runs in time polynomial in the size of x and x has a solution if and only if y has a solution, *i.e.*, $x \in \Pi$ if and only if $y \in \Pi'$.

Definition 1. *A* **parameterized problem** $\Pi \subseteq \Sigma^* \times \Sigma^*$ *has instances of the form* (x, y)*, where* x *is called the* **main part** *and* y *is called the* **parameter**.

Definition 2. *A parameterized problem* Π *is* **fixed-parameter tractable** *if there exists an algorithm* A *to determine if instance* (x, y) *is in* Π *in time* $f(y) \cdot |x|^{\alpha}$*, where* $f : \Sigma^+ \mapsto \mathcal{N}$ *is an arbitrary function and* α *is a constant independent of* x *and* y.

Given a decision problem Π with a parameter p, let $\langle p \rangle$-Π denote the parameterized problem associated with Π that is based on parameter p and let $\langle p_c \rangle$-Π denote the subproblem of $\langle p \rangle$-Π in which p has value c for some constant $c \geq 0$. One can establish that a parameterized problem Π is not fixed-parameter tractable by using a parametric reduction[2] to show that Π is hard for any of the classes of the **W-hierarchy** $= \{FPT, W[1], W[2], \ldots, W[P], XP\}$ except FPT, where FPT is the class of fixed-parameter tractable parameterized problems (see [4] for details). These classes are related as follows:

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[P] \subseteq \cdots \subseteq XP$$

If a parameterized problem is $\mathcal{C}$-hard for any class $\mathcal{C}$ in the W-hierarchy above FPT then that problem is not fixed-parameter tractable unless $FPT = \mathcal{C}$.

The following lemmas will be used in the analyses given in the next section.

Lemma 3. [16, Lemma 2.1.25] *Given decision problems* Π *and* Π' *with parameters* p *and* p'*, respectively, if* $\Pi \leq_m \Pi'$ *such that* $p' = g(p)$ *for an arbitrary function* g*, then* $\langle p \rangle$-Π *parametrically reduces to* $\langle p' \rangle$-Π'.

Lemma 4. [16, Lemma 2.1.35] *Given a set* S *of aspects of a decision problem* Π*, if* Π *is* NP*-hard when the value of every aspect* $s \in S$ *is fixed, then the parameterized problem* $\langle S \rangle$-Π *is not in* XP *unless* $P = NP$.

3 Results

The analyses in this section will focus on the following three problems:

Bounded DFA Intersection (BDFAI)
Instance: A set A of DFA over an alphabet Σ and a positive integer k.
Question: Is there a string $x \in \Sigma^k$ that is accepted by each DFA in A?

i/o-deterministic FST Intersection (FST-I)
Instance: A set A of i/o-deterministic FST, all of whose input and output alphabets are Σ_i and Σ_o, respectively, and a string $u \in \Sigma_i^+$.
Question: Is there a string $s \in \Sigma_o^{|u|}$ such that the string-pair u/s is accepted by each FST in A?

[2] Given parameterized problems Π and Π', Π **parametrically reduces to** Π' if there is an algorithm A that transforms an instance (x, y) of Π into an instance (x', y') of Π' such that A runs in time $f(y)|x|^{\alpha}$ time for an arbitrary function f and a constant α independent of both x and y, $y' = g(y)$ for some arbitrary function g, and $(x, y) \in \Pi$ if and only if $(x', y') \in \Pi'$.

i/o-DETERMINISTIC FST COMPOSITION (FST-C)
Instance: A set A of i/o-deterministic FST, all of whose input and output alphabets are Σ, a composition-order O on these FST, and a string $u \in \Sigma^+$.
Question: Is there a sequence of strings $\{s_0, s_1, \ldots, s_{|A|}\}$ with $s_0 = u$ and $s_i \in \Sigma^{|u|}$ for $1 \leq i \leq |A|$ such that for the ordering $\{a_1, a_2, \ldots, a_{|A|}\}$ of A under O and $1 \leq i \leq |A|$, s_{i-1}/s_i is accepted by a_i?

The version of problem BDFAI in which $x \in \Sigma^*$ is $PSPACE$-complete [9] and problem FST-I is NP-hard by a slight modification of the reduction given in [2, Section 5.5.1]. All parameterized complexity analyses given in this section will be done relative to the following aspects: the number of finite-state automata in A ($|A|$), the required length of the result-string (k in the case of BDFAI, $|u|$ in the case of FST-I and FST-C), the maximum number of states of any finite-state automaton in A ($|Q|$), and the size of the alphabet ($|\Sigma|$ in the case of BDFAI and FST-C, $|\Sigma_i|$ *and* $|\Sigma_o|$ in the case of FST-I).

3.1 Bounded DFA Intersection

Hardness results will be derived via reductions from the following problems:

LONGEST COMMON SUBSEQUENCE (LCS) [5, Problem SR10]
Instance: A set of strings $X = \{x_1, \ldots, x_k\}$ over an alphabet Σ, an integer m.
Question: Is there a string $y \in \Sigma^m$ that is a subsequence of x_i for $i = 1, \ldots, k$?

DOMINATING SET [5, Problem GT2]
Instance: A graph $G = (V, E)$, an integer k.
Question: Is there a set of vertices $V' \subseteq V$, $|V'| \leq k$, such that each vertex in V is either in V' or adjacent to a vertex in V'?

Note that all reductions below are phrased in terms of BDFAI_R, the restricted version of BDFAI in which $k \leq |Q|$.

Lemma 5. LCS $\leq_m$ BDFAI_R.

Proof. Given an instance $\langle X, k, \Sigma, m\rangle$ of LCS, construct the following instance $\langle A', \Sigma', k'\rangle$ of BDFAI_R: Let $\Sigma' = \Sigma$ $k' = m$, and A' be created by applying to each string $x \in X$ the algorithm in [1] which produces a DFA on $|x|+1$ states that recognizes all subsequences of a string x. Note that in the constructed instance of BDFAI_R, $|A'| = k$, $k' = m$, and $|\Sigma'| = |\Sigma|$; moreover, $k' = m < |Q|$. □

Lemma 6. DOMINATING SET $\leq_m$ BDFAI_R.

Proof. Given an instance $\langle G = (V, E), k\rangle$ of DOMINATING SET, construct the following instance $\langle A', \Sigma', k'\rangle$ of BDFAI_R: Let $\Sigma' = V$ be an alphabet such that each vertex $v \in V$ has a distinct corresponding symbol in Σ', and let $k' = k$. For each $v \in V$, let $adj(v)$ be the set of vertices in V that are adjacent to v in G (including v itself) and $nonadj(v) = V - adj(v)$. For each vertex $v \in V$, construct a two-state DFA $A_v = \langle\{q_1, q_2\}, \Sigma', \delta, q_1, \{q_2\}\rangle$ with transition relation

$\delta = \{(q_1, v', q_1) \mid v' \in nonadj(v)\} \cup \{(q_1, v', q_2) \mid v' \in adj(v)\} \cup \{(q_2, v', q_2) \mid v' \in V\}$. Let A' be the set consisting of all DFA A_v corresponding to vertices $v \in V$ plus the $k+1$ state DFA that recognizes all strings in ${\Sigma'}^k$. Note that in the constructed instance of BDFAI$_R$, $|A'| = |\Sigma'| + 1 = |V| + 1$, $k' = k$, and $|Q| = k + 1$; moreover, $k' = k \leq |V|$ and $k' = k < |Q|$. □

Lemma 7. BDFAI$_R$ $\leq_m$ BDFAI$_R$ *such that* $|\Sigma| = 2$.

Proof. Given an instance $\langle A, \Sigma, k\rangle$ of BDFAI$_R$, construct the following instance $\langle A', \Sigma', k'\rangle$ of BDFAI$_R$: Let $\Sigma' = \{0, 1\}$ and assign each symbol in Σ a binary codeword of fixed length $\ell = \lceil \log |\Sigma| \rceil$. For each DFA $a \in A$, create a DFA $a' \in A'$ by adjusting Q and δ such that each state q and its outgoing transitions in a is replaced with a "decoding tree" on $2^\ell - 1$ states in a' that uses ℓ bits to connect q to the appropriate states. Finally, let $k' = k\ell$. Note that in the constructed instance of BDFAI$_R$, $|A'| = |A|$ and $|\Sigma'| = 2$; moreover, as $(|Q| + 1)(|\Sigma| - 1) \leq |Q'|$ and $k \leq |Q|$, $k' = k\ell \leq |Q|\lceil \log |\Sigma| \rceil \leq (|Q| + 1)(|\Sigma| - 1) \leq |Q'|$. □

Theorem 8.

1. BDFAI$_R$ *is NP-hard when* $|\Sigma| = 2$.
2. $\langle k, |\Sigma|\rangle$-BDFAI$_R$ *is in FPT.*
3. $\langle |A|, |Q|\rangle$-BDFAI$_R$ *is in FPT.*
4. $\langle |Q|, |\Sigma|\rangle$-BDFAI$_R$ *is in FPT.*
5. $\langle |A|, k\rangle$-BDFAI$_R$ *is W*[1]*-hard.*
6. $\langle k, |Q|\rangle$-BDFAI$_R$ *is W*[2]*-hard.*
7. $\langle |A|, |\Sigma|_2\rangle$-BDFAI$_R$ *is W*[t]*-hard for all* $t \geq 1$.
8. $\langle |\Sigma|\rangle$-BDFAI$_R \notin XP$ *unless* $P = NP$.

Proof of (1): Follows from the NP-hardness of LCS [5, Problem SR10], the reduction in Lemma 5 from LCS to BDFAI$_R$, and the reduction in Lemma 7 from BDFAI$_R$ to BDFAI$_R$ in which $|\Sigma| = 2$.
Proof of (2): Follows from the algorithm that generates all $|\Sigma|^k$ possible k-length strings over alphabet Σ and checks each string in $O(|A|k)$ time to see whether that string is accepted by each of the DFA in A. The algorithm as a whole runs in $O(|\Sigma|^k k|A|)$ time, which is fixed-parameter tractable relative to k and $|\Sigma|$.
Proof of (3): Follows from the algorithm that constructs the intersection DFA of all DFA in A and the $k+1$-state DFA that recognizes all strings in Σ^k, and then applies depth-first search to the transition diagram for this intersection DFA to determine if any of its final states are reachable from its start state. The intersection DFA can be created in $O(|Q|^{|A|+1}(k+1)|\Sigma|^2) = O(|Q|^{|A|+1}2k|\Sigma|^2) = O(|Q|^{|A|+1}k|\Sigma|^2)$ time. As the graph $G = (V, E)$ associated with the transition diagram of this DFA has $|V| \leq (k+1)|Q|^{|A|} \leq 2k|Q|^{|A|}$ states and $|A| \leq (k+1)|Q|^{|A|}|\Sigma| \leq 2k|Q|^{|A|}|\Sigma|$ arcs and depth-first search runs in $O(|V| + |A|)$ time, the algorithm as a whole runs in $O(|Q|^{|A|+1}k|\Sigma|^2)$ time, which is fixed-parameter tractable relative to $|A|$ and $|Q|$.
Proof of (4): This result follows from the observation that there are at most $|Q|^{|\Sigma||Q|} \times 2^{|Q|} \leq |Q|^{2|\Sigma||Q|}$ i/o-deterministic FST for any choice of $|Q|$ and

$|\Sigma|$. Hence, the number of different FST in any set A is at most $|Q|^{2|\Sigma||Q|+1}$. This suggests the algorithm that removes all redundant FST in A and then performs the algorithm given in part (3) above. The first step involves checking the isomorphism of the transition diagrams of all FST in A, where each transition diagram has at most $|Q|$ vertices and at most $|Q|^2|\Sigma|$ edges, which can be done in $O(((|A|(|A|-1)/2)|Q|^{|Q|}|Q|^2|\Sigma|) = O((|A|^2|Q|^{|Q|+2}|\Sigma|)$ time. Hence, the algorithm as a whole runs in $O(|Q|^{(|Q|^2|\Sigma||Q|)+1}k|\Sigma^2||A|^2)$ time, which is fixed-parameter tractable relative to $|Q|$ and $|\Sigma|$.

Proof of (5): Follows from the $W[1]$-completeness of $\langle k, m\rangle$-LCS [3], the reduction in Lemma 5 from LCS to BDFAI_R in which $|A'| = k$ and $k' = m$, and Lemma 3.

Proof of (6): Follows from the $W[2]$-completeness of $\langle k\rangle$-DOMINATING SET [4], the reduction in Lemma 6 from DOMINATING SET to BDFAI_R in which $k' = k$ and $|Q| = k + 1$, and Lemma 3.

Proof of (7): Follows from the $W[t]$-hardness of $\langle k\rangle$-LCS for $t \geq 1$ [3], the reduction in Lemma 5 from LCS to BDFAI_R in which $|A'| = k$, the reduction in Lemma 7 from BDFAI_R to BDFAI_R in which $|A'| = |A|$ and $|\Sigma'| = 2$, and Lemma 3.

Proof of (8): Follow from (1) and Lemma 4. □

As BDFAI_R is a restriction of BDFAI, all hardness results above also hold for BDFAI. However, as the value of k is not necessarily bounded by a polynomial in the instance size in BDFAI, the algorithm for part (4) only works (and hence results (4) and (5) only hold) for BDFAI if k is also included in the parameter.

3.2 *i/o*-Deterministic FST Intersection

Lemma 9. $\text{BDFAI}_R \leq_m$ FST-I.

Proof. Given an instance $\langle A, \Sigma, k\rangle$ of BDFAI_R, construct the following instance $\langle A', \Sigma_i', \Sigma_o', u'\rangle$ of FST-I: Let $\Sigma_i' = \{\Delta\}$ for some symbol $\Delta \notin \Sigma$, $\Sigma_o' = \Sigma$ and $u' = \Delta^k$. Given a DFA $a = \langle Q, \Sigma, \delta, s, F\rangle$, let $FST_u(a) = \langle Q, \Sigma_i', \Sigma, \delta_F, s, F\rangle$ be the FST such that $\delta_F = \{(q, \Delta, x, q') \mid (q, x, q') \in \delta\}$ and let A' be the set consisting of all FST $FST_u(a)$ corresponding to DFA $a \in A$. Note that in the constructed instance of FST-I, $|A'| = |A|$, $|u'| = k$, $|Q'| = |Q|$, $|\Sigma_u'| = 1$, and $|\Sigma_s'| = |\Sigma|$. □

Theorem 10.

1. FST-I *is NP-hard when* $|\Sigma_i| = 1$ *and* $|\Sigma_o| = 2$ *and when* $|Q| = 4$ *and* $|\Sigma_o| = 3$.
2. $\langle|u|, |\Sigma_o|\rangle$-FST-I, $\langle|A|, |Q|\rangle$-FST-I, *and* $\langle|Q|, |\Sigma|\rangle$-FST-I *are in FPT.*
3. $\langle|A|, |u|, |\Sigma_i|_1\rangle$-FST-I *is* $W[1]$*-hard.*
4. $\langle|u|, |Q|, |\Sigma_i|_1\rangle$-FST-I *is* $W[2]$*-hard.*
5. $\langle|A|, |\Sigma_i|_1, |\Sigma_o|_2\rangle$-FST-I *is* $W[t]$*-hard for all* $t \geq 1$.
6. $\langle|\Sigma_o|, |\Sigma_i|\rangle$-FST-I *and* $\langle|Q|, |\Sigma_o|\rangle$-FST-I *are not in XP unless* $P = NP$.

Proof. (*Sketch*) Almost all results follow by arguments similar to those in Theorem 8 relative to the reduction in Lemma 9 and the appropriate results in Theorem 8. The NP-hardness of FST-I when $|Q| = 4$ and $|\Sigma_o| = 3$ follows from the reduction in [2, Section 5.5.3]. □

3.3 i/o-Deterministic FST Composition

The following reduction formalizes the observation (made independently by Karttunen [8]) that FSA intersection can be simulated by the composition of identity-relation FST.

Lemma 11. BDFAI$_R$ $\leq_m$ FST-C.

Proof. Given an instance $\langle A, \Sigma, k\rangle$ of BDFAI$_R$, construct the following instance $\langle A', O', \Sigma', u', \rangle$ of FST-C: Let $\Sigma' = \Sigma \cup \{\Delta\}$ for some symbol $\Delta \notin \Sigma$, and $u' = \Delta^k$. Given a DFA $a = \langle Q, \Sigma, \delta, s, F\rangle$, let $FST_u(a) = \langle Q, \Sigma, \Sigma, \delta_F, s, F\rangle$ be the FST such that $\delta_F = \{(q, x, x, q') \mid (q, x, q') \in \delta\}$. Let A' be the set consisting of all FST $FST_u(a)$ corresponding to DFA $a \in A$ plus the special FST $FST_{init} = \langle\{q_1\}, \{\Delta\}, \Sigma, \delta, q_1, \{q_1\}\rangle$ for which $\delta = \{(q_1, \Delta, x, q') \mid x \in \Sigma\}$, and let O' be an ordering on A' such that FST_{init} is the first FST in O and the other FST appear in an arbitrary order. Note that in the constructed instance of FST-C, $|A'| = |A|+1$, $|u'| = k$, $|Q'| = \max(|Q|, 1) = |Q|$, and $|\Sigma'| = |\Sigma|+1$. □

Theorem 12.

1. FST-C *is NP-hard when* $|\Sigma| = 3$.
2. $\langle|u|, |\Sigma|\rangle$-FST-C *and* $\langle|A|, |Q|\rangle$-FST-C *are in FPT.*
3. $\langle|A|, |u|\rangle$-FST-C *is W*[1]*-hard.*
4. $\langle|u|, |Q|\rangle$-FST-C *is W*[2]*-hard.*
5. $\langle|A|, |\Sigma|_3\rangle$-FST-C *is W*[$t$]*-hard for all* $t \geq 1$.
6. $\langle|\Sigma|\rangle$-FST-C *is not in XP unless* $P = NP$.

Proof. (*Sketch*) Almost all results follow by arguments similar to those in Theorem 8 relative to the reduction in Lemma 11 and the appropriate results in Theorem 8. The fixed-parameter tractability of $\langle|u|, |\Sigma|\rangle$-FST-C follows by a variant of an algorithm in [13, Theorem 4.3.3, Part (3)] that uses an $|\Sigma|^{|u|}$-length bit-vector to store the intermediate sets of strings produced during the FST composition. □

4 Discussion

All parameterized complexity results for problems BDFAI, FST-I, and FST-C that are either stated or implicit in the previous section are shown in Tables 1 and 2. Recall that results for problems FST-I and FST-C are stated relative to restricted FST; hence, only hardness results necessarily hold for these problems relative to general FST, and given FPT algorithms are at best outlines for possible FPT algorithms for general FST (see [13, Sections 4.3.3 and 4.4.3] for further discussion). Future research should both look for algorithms that exploit the sets of aspects underlying the state Cartesian product ($|A|$, $|Q|$) and exhaustive string generation ($|u|$, $|\Sigma|$) constructions) in new ways and consider other aspects of finite-state automaton intersection and composition problems, such as characterizations of logic formulas describing the automata [12].

Table 1. The Parameterized Complexity of the BOUNDED DFA INTERSECTION and i/o-DETERMINISTIC FST COMPOSITION Problems. (a) The BOUNDED DFA INTERSECTION Problem. (b) The i/o-DETERMINISTIC FST COMPOSITION Problem.

(a)

Parameter	Alphabet Size $\|\Sigma\|$	
	Unbounded	Parameter
–	NP-hard	$\notin XP$ unless $P = NP$
$\|A\|$	$W[t]$-hard	$W[t]$-hard
k	$W[2]$-hard	FPT
$\|Q\|$	$W[2]$-hard	???
$\|A\|$, k	$W[1]$-hard	FPT
$\|A\|$, $\|Q\|$	???	???
k, $\|Q\|$	$W[2]$-hard	FPT
$\|A\|$, k, $\|Q\|$	FPT	FPT

(b)

Parameter	Alphabet Size $\|\Sigma\|$	
	Unbounded	Parameter
–	NP-hard	$\notin XP$ unless $P = NP$
$\|A\|$	$W[t]$-hard	$W[t]$-hard
$\|u\|$	$W[2]$-hard	FPT
$\|Q\|$	$W[2]$-hard	???
$\|A\|$, $\|u\|$	$W[1]$-hard	FPT
$\|A\|$, $\|Q\|$	FPT	FPT
$\|u\|$, $\|Q\|$	$W[2]$-hard	FPT
$\|A\|$, $\|u\|$ $\|Q\|$	FPT	FPT

Table 2. The Parameterized Complexity of the i/o-DETERMINISTIC FST INTERSECTION Problem.

Parameter	Alphabet Sizes ($\|\Sigma_i\|$,$\|\Sigma_o\|$)			
	(Unb,Unb)	(Unb,Prm)	(Prm,Unb)	(Prm,Prm)
–	NP-hard	$\notin XP$ unless $P = NP$	$\notin XP$ unless $P = NP$	$\notin XP$ unless $P = NP$
$\|A\|$	$W[t]$-hard	$W[t]$-hard	$W[t]$-hard	$W[t]$-hard
$\|u\|$	$W[2]$-hard	FPT	$W[2]$-hard	FPT
$\|Q\|$	$W[2]$-hard	$\notin XP$ unless $P = NP$	$W[2]$-hard	FPT
$\|A\|, \|u\|$	$W[1]$-hard	FPT	$W[1]$-hard	FPT
$\|A\|, \|Q\|$	FPT	FPT	FPT	FPT
$\|u\|, \|Q\|$	$W[2]$-hard	FPT	$W[2]$-hard	FPT
$\|A\|, \|u\|, \|Q\|$	FPT	FPT	FPT	FPT

One such aspect of great interest is FST ambiguity (essentially, the maximum number of strings associated with any input string by a FST). Problems FST-I and FST-C are solvable in low-order polynomial time when they are restricted to operate on sequential FST, *i.e.*, FST that associate at most one output string with any input string. The reduction in Lemma 11 suggests that the presence of only one i/o-deterministic FST can make FST composition NP-hard. What about more restricted classes of FST? One candidate is the **p-subsequential FST** [11], which are essentially sequential FST which are allowed to append any one of a fixed set of p strings to their output. Such FST seem adequate for efficiently representing the ambiguity present in many applications [11]. However, the following result suggests that this observed efficiency is not universal.

Theorem 13. 2-SUBSEQUENTIAL FST COMPOSITION *is NP-hard.*

Proof. (*Sketch*) Given instances of BDFAI_R created in Lemma 7, the reduction in Lemma 11 can be modified by setting $u' = \Delta$ and replacing the i/o-deterministic FST FST_{init} with a set of $|u|$ 2-sequential FST that echo their input and append a 0 or a 1 (that is, a set of 2-subsequential FST whose composition generates all possible strings of length $|u|$ over the alphabet $\{0, 1\}$). □

Acknowledgments. The author would like to thank the CIAA referees for various helpful suggestions, and for pointing out several errors in the original manuscript as well as solutions for some of these errors.

References

1. R.A. Baeza-Yates. 1991. Searching Subsequences. *Theoretical Computer Science*, 78, 363–376.
2. G.E. Barton, R.C. Berwick, and E.S. Ristad. 1987. *Computational Complexity and Natural Language*. MIT Press, Cambridge, MA.
3. H.L. Bodlaender, R.G. Downey, M.R. Fellows, and H.T. Wareham. 1995. The Parameterized Complexity of Sequence Alignment and Consensus. *Theoretical Computer Science*, 147(1–2), 31–54.
4. R.G. Downey and M.R. Fellows. 1999. *Parameterized Complexity*. Springer-Verlag, Berlin.
5. M.R. Garey and D.S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco.
6. J.E. Hopcroft and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.
7. R.M. Kaplan and M. Kay. 1994. Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20(3), 331–378.
8. L. Karttunen. 1998. The Proper Treatment of Optimality in Computational Phonology. Technical Report ROA-258-0498, Rutgers Optimality Archive.
9. D. Kozen. 1977. Lower Bounds for Natural Proof Systems. In *18th IEEE Symposium on Foundations of Computer Science*, pp. 254-266. IEEE Press.
10. R.P. Kurshan. 1994. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press.
11. M. Mohri. 1997. On the Use of Sequential Transducers in Natural Language Processing. In E. Roche and Y. Schabes, eds., *Finite-State Language Processing*, pp. 355–382. MIT Press, Cambridge, MA.
12. H. Straubing. 1994. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhäuser, Boston, MA.
13. H.T. Wareham. 1999. *Systematic Parameterized Complexity Analysis in Computational Phonology*. Ph.D. thesis, Department of Computer Science, University of Victoria. Technical Report ROA-318-0599, Rutgers Optimality Archive.

Directly Constructing Minimal DFAs: Combining Two Algorithms by Brzozowski

Bruce W. Watson

University of Pretoria
(Department of Computer Science)
Pretoria 0002, South Africa
watson@OpenFIRE.org
www.OpenFIRE.org

Abstract. In this paper, we combine (and refine) two of Brzozowski's algorithms — yielding a single algorithm which constructs a minimal deterministic finite automaton (DFA) from a regular expression.

1 Introduction

To obtain a minimal DFA, an implementor usually codes separate construction and minimization algorithms, composing them (at run-time) to yield the minimal DFA. A single, combined algorithm, which both constructs the DFA and simultaneously minimizes it is likely to be even more efficient thanks to the fact that fewer intermediate data-structures are built. Recent examples of this phenomenon can be found in [3,4,8,9,11,13] which all present algorithms that construct automata while maintaining minimality or near minimality. Those algorithms have been shown to out-perform the naïve run-time composition of a construction algorithm with a minimization algorithm.

In this paper, the two algorithm design focii are:

1. The real-life performance of the algorithm. The asymptotic running time is usually a complex function of the inherent complexity of the input regular expression, which we do not discuss further here.
2. The quality of the output automaton — in this case, the result is a *minimal* automaton.

1.1 Related Work

There are numerous possible minimizing DFA construction algorithms, at least one for each possible combination of a construction algorithm with a minimization algorithm. (Using [10] as a basis, this yields at least two hundred possibilities.) The algorithm presented here is a relatively elegant combination of two algorithms which themselves are simple and easily presented. The resulting algorithm is easy to manipulate and refine, and therefore easy to optimize and implement. Preliminary benchmarking indicates that the new algorithm outperforms a runtime composition of the two component algorithms.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 311–317, 2001.

1.2 Preliminaries

We assume that the reader is reasonably familiar with elementary finite automata theory, definitions and principles, including: states, transitions, determinism, minimality, regular expressions, language denoted by a regular expression, derivatives (of a regular expression), and *similarity* of regular expressions. (In the remainder of this paper, all derivatives are actually a short-hand for their similarity equivalence classes.) Readers who are not familiar with this material should consult any one of the standard text-books, for example [7,14].

One definition which is not always presented in text-books is that of *reversal*. We can define function[1] reverse which reverses:

- a string, returning a string with the letters of the original string in the reverse order;
- a language, by returning the language consisting of the reversal of each string in the original language;
- a regular expression; this can be defined inductively on the structure of regular expressions; and
- a finite automaton, by returning a new automaton in which the direction (source and destination) of each transition is reversed, start states (in the original automaton) are made into final states, and final states (in the original automaton) are made into start states.

We define subset to be the *subset construction* — a function which takes a finite automaton and returns a DFA with no unreachable states, accepting the same language (see, for example, [7]).

All of the algorithms presented here are in the guarded command language (with some additional annotations for comments), a type of pseudo-code — see [5].

2 The Component Algorithms

We begin with the construction portion of such an algorithm. From [10, Chapter 6], there are at least twenty known construction algorithms; unfortunately, the performance data presented in [10, Chapter 14] does not include all of those algorithms. Nonetheless, annecdotal experience (also from industrial applications such as in computation linguistics) indicates that a good choice is the *derivatives*-based algorithm by Brzozowski [2] (we refer to this algorithm as Brzconstr). That algorithm, which yields a DFA, is exceptionally easy to present and to implement, satisfying the first of our two goals. Furthermore, it can be efficiently implemented[2] by providing a regular expression *simplifier*, since each state (in

[1] Despite being called a function, it would likely be implemented as an imperative program.

[2] Extensive benchmarking data for this is not available, however, industry applications of the algorithm have show that it is more efficient that other popular algorithms such as the Aho-Sethi-Ullman algorithm [10].

the resulting automaton) is represented by a regular expression. Such a simplifier is then able to identify two states which may otherwise have been distinguished, for example, the states $a \cdot b$ and $a \cdot \varepsilon \cdot b$ would be identified if the simplifier is aware of the identity $E \cdot \varepsilon \equiv E$. (In order for Brzozowski's construction to terminate (correctly), the simplifier must at least recognize similarity; we do not discuss this further in this paper.)

We now focus on the selection of a minimization algorithm. There are several known minimization algorithms — see [10, Chapter 7]. Presently, Hopcroft's algorithm is the algorithm with the best-known asymptotic running time (of $\mathcal{O}(n \log n)$, for n the number of states). Unfortunately, that algorithm is also one of the more difficult algorithms to present, manipulate, refine, and implement — cf. that David Gries's paper [6] shed significant light on the algorithm's derivation. As we discuss in [12], Brzozowski's minimization algorithm [1] has also proven to be exceptionally good in practice, usually out-performing Hopcroft's algorithm. (For a performance comparison, see [10, Chapter 15].) Without presenting the details (which can be found in [10]), Brzozowski's minimization algorithm is defined as follows (for automaton M, where the semicolon is used for sequential composition of the functions, and we may choose to implement the 'functions' as imperative programs):

$$\mathsf{Brzmin}(M) = (\mathsf{reverse}; \mathsf{subset}; \mathsf{reverse}; \mathsf{subset})(M)$$

3 Combining Algorithms

In this section, we combine the two algorithms by Brzozowski to yield a more efficient single algorithm than their run-time composition. Initially, we consider expression:

$$(\mathsf{Brzconstr}; \mathsf{Brzmin})(E)$$

Expanding the definition of Brzmin, we get

$$(\mathsf{Brzconstr}; \mathsf{reverse}; \mathsf{subset}; \mathsf{reverse}; \mathsf{subset})(E)$$

Straight-forward refinements and improvements of this algorithm can be obtained by combining algorithm components which are adjacent in terms of composition, and for the moment we ignore the right-most component (the subset).

We can make an important observation at this point: reversal commutes with all construction algorithms (mapping a regular expression to a finite automaton). This allows us to switch the first two components of the composition:

$$(\mathsf{reverse}; \mathsf{Brzconstr}; \mathsf{subset}; \mathsf{reverse}; \mathsf{subset})(E)$$

The latter half of Brzmin (reverse; subset) only requires that its input is a DFA accepting the language denoted by reverse(E). Since Brzconstr already yields a DFA, the invocation of subset following Brzconstr is redundant, and we simplify it to

$$(\mathsf{reverse}; \mathsf{Brzconstr}; \mathsf{reverse}; \mathsf{subset})(E)$$

To make further improvements, we switch to the imperative version of the algorithm. (Q refers to the set of states $S, F \subseteq Q$ are the sets (respectively) of start and final states, δ is the transition relation, and $\mathcal{L}$ is an overloaded function giving the language of a regular expression or finite automaton.)

Algorithm 3.1:

$E' := \mathsf{reverse}(E)$;
$Q, \delta, S, F := \emptyset, \emptyset, \{E'\}, \emptyset$;
$done, to\_do := \emptyset, \{E'\}$;
do $to\_do \neq \emptyset \rightarrow$
  let p be some state such that $p \in to\_do$;
  $done, to\_do := done \cup \{p\}, to\_do \setminus \{p\}$;
  $destination := a^{-1}p$ — the left derivative of p by a;
  if $destination \notin done \rightarrow$
    { $destination$'s out-transitions are still to be built }
    $to\_do := to\_do \cup \{destination\}$
  [] $destination \in done \rightarrow$ **skip**
  fi;
  $Q := Q \cup \{destination\}$;
  { make a transition from p to $destination$ on a }
  $\delta(p, a) := destination$;
  if $\varepsilon \in \mathcal{L}(destination) \rightarrow$
    { this should be a final state }
    $F := F \cup \{destination\}$
  [] $\varepsilon \notin \mathcal{L}(destination) \rightarrow$ **skip**
  fi
od;
{ $\mathcal{L}(Q, \delta, S, F) = \mathcal{L}(\mathsf{reverse}(E))$ }
$(Q, \delta, S, F) := \mathsf{subset}(\mathsf{reverse}(Q, \delta, S, F))$;
{ $\mathcal{L}(Q, \delta, S, F) = \mathcal{L}(E)$ }

□

In the above algorithm, we begin by reversing the regular expression E (giving E'). Thanks to the symmetry of derivatives, we have an alternative: *use right derivatives* instead of left derivatives. This yields the following algorithm (in which we have removed E' and the changes from the previous algorithm have been underlined):

Algorithm 3.2:

$Q, \delta, S, F := \emptyset, \emptyset, \{E\}, \emptyset$;
$done, to\_do := \emptyset, \{E\}$;
do $to\_do \neq \emptyset \rightarrow$
  let p be some state such that $p \in to\_do$;
  $done, to\_do := done \cup \{p\}, to\_do \setminus \{p\}$;

destination : = pa^{-1} — the right derivative of p by a;
if *destination* $\notin$ *done* $\rightarrow$
{ *destination*'s out-transitions are still to be built }
to_do : = *to_do* $\cup$ {*destination*}
$[\!]$ *destination* $\in$ *done* $\rightarrow$ **skip**
fi;
Q : = $Q \cup$ {*destination*};
{ make a transition from p to *destination* on a }
$\delta(p, a)$: = *destination*;
if $\varepsilon \in \mathcal{L}$(*destination*) $\rightarrow$
{ this should be a final state }
F : = $F \cup$ {*destination*}
$[\!]$ $\varepsilon \notin \mathcal{L}$(*destination*) $\rightarrow$ **skip**
fi
od;
{ $\mathcal{L}(Q, \delta, S, F) = \mathcal{L}(\mathsf{reverse}(E))$ }
(Q, δ, S, F) : = $\mathsf{subset}(\mathsf{reverse}(Q, \delta, S, F))$;
{ $\mathcal{L}(Q, \delta, S, F) = \mathcal{L}(E)$ }

□

Our remaining goal is to combine the final reversal (of the DFA) with the preceding portion of the algorithm. Finite automaton reversal is as simple as exchanging the places of the start and final states (S and F), and reversing the update of the transition function (δ), yielding:

Algorithm 3.3:

Q, δ, S, F : = $\emptyset, \emptyset, \{E\}, \emptyset$;
done, *to_do* : = $\emptyset, \{E\}$;
do *to_do* $\neq \emptyset \rightarrow$
let p be some state such that $p \in$ *to_do*;
done, *to_do* : = *done* $\cup \{p\}$, *to_do* $\setminus \{p\}$;
destination : = pa^{-1} — the right derivative of p by a;
if *destination* $\notin$ *done* $\rightarrow$
{ *destination*'s out-transitions are still to be built }
to_do : = *to_do* $\cup$ {*destination*}
$[\!]$ *destination* $\in$ *done* $\rightarrow$ **skip**
fi;
Q : = $Q \cup$ {*destination*};
{ make a transition from *destination* to p on a }
δ(*destination*, a) : = p;
if $\varepsilon \in \mathcal{L}$(*destination*) $\rightarrow$
{ this should be a final state }
F : = $F \cup$ {*destination*}
$[\!]$ $\varepsilon \notin \mathcal{L}$(*destination*) $\rightarrow$ **skip**

fi
od;
$\{\ \mathcal{L}(Q,\delta,S,F) = \mathcal{L}(\mathsf{reverse}(E))\ \}$
$(Q,\delta,S,F) := \mathsf{subset}(Q,\delta,\underline{F,S})$;
$\{\ \mathcal{L}(Q,\delta,S,F) = \mathcal{L}(E)\ \}$

□

This is our final algorithm. We will not manipulate the remaining composition further (since it does not appear to yield any performance or readability advantages), nor will we explicitly present subset.

4 Closing Comments

The derived algorithm has the following characteristics:

- The derivation of the algorithm is easily understood and the correctness is easily established.
- Thanks to the easily-understood derivation, the algorithm is also quickly implemented.
- The final algorithm is as easy to implement as the original Brzconstr. The differences can be factored, leaving a common algorithmic skeleton. (This technique is used in [10] for keyword pattern matching algorithms with a common skeleton.)
- One of the algorithmic components, the subset construction, is usually already implemented in automata toolkits.
- Like Brzozowski's derivatives-based construction algorithm, the performance is easily improved by implementing an improved regular expression simplifier.
- The component algorithms have excellent performance in practice. It follows that the new algorithm will display similar (if not better) performance — though this is still being verified.

Acknowledgements. I would like to thank Nanette Saes and the anonymous referees for improving the quality of this paper.

References

1. Brzozowski, J.A. Canonical regular expressions and minimal state graphs for definite events, in: *Mathematical Theory of Automata.* (pp. 529–561, Vol. 12, MRI Symposia Series, Polytechnic Press, Polytechnic Institute of Brooklyn, NY, 1962).
2. Brzozowski, J.A. Derivatives of regular expressions. (J. ACM 11(4), pp. 481–494, 1964).
3. Daciuk, J.D., Watson, B.W. and R.E. Watson. An Incremental Algorithm for Constructing Acyclic Deterministic Transducers. (Proceedings of the International Workshop on Finite State Methods in Natural Language Processing, Ankara, Turkey, 30 June–1 July 1998).

4. Daciuk, J.D., Mihov, S., Watson, B.W. and R.E. Watson. Incremental Construction of Minimal Acyclic Finite State Automata. (to appear in *Computational Linguistics*, 2000).
5. Dijkstra, E.W. *A Discipline of Programming.* (Prentice Hall, Englewood Cliffs, N.J., 1976).
6. Gries, D. Describing an Algorithm by Hopcroft. (Acta Informatica 2, pp. 97–109, 1973).
7. Hopcroft, J.E. and J.D. Ullman. *Introduction to Automata, Theory, Languages and Computation.* (Addison-Wesley, Reading, M.A., 1979).
8. Mihov, S. Direct Building of Minimal Automaton for Given List. (Available from `stoyan@lml.acad.bg`).
9. Revuz, D. Minimisation of Acyclic Deterministic Automata in Linear Time. (Theoretical Computer Science 92, pp. 181-189, 1992).
10. Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995). See `www.OpenFIRE.org`
11. Watson, B.W. A Fast New Semi-Incremental Algorithm for the Construction of Minimal Acyclic DFAs. (Proceedings of the Third Workshop on Implementing Automata, Rouen, France, September 1998).
12. Watson, B.W. A History of Brzozowski's DFA minimization algorithm. (Poster at the International Conference on Implementing Automata, London, Ontario, 2000).
13. Watson, B.W. A New Recursive Incremental Algorithm for Building Minimal Acyclic Deterministic Finite Automata. (to appear, 2000).
14. Wood, D. *Theory of Computation.* (Harper & Row, New York, N.Y., 1987).

The MERLin Environment Applied to ⋆-NFAs

Lynette van Zijl*, John-Paul Harper, and Frank Olivier

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

Abstract. There are known mechanisms to succinctly describe regular languages, such as nondeterministic finite automata, boolean automata, and statecharts. The MERLin project is an investigation into and comparison of different description mechanisms for the regular languages. In particular, we are concerned with descriptions which, for a specific application domain, often achieve succinctness. To this end we implemented a **M**odelling **E**nvironment for **R**egular **L**anguages (MERLin). This paper describes the application of the MERLin system to analyze the behaviour of *selective nondeterministic finite automata*.

1 Introduction

There are known mechanisms to succinctly describe regular languages, such as nondeterministic finite automata (NFAs) [12], boolean automata [10] and statecharts [5]. However, in practical applications, it may happen that the theoretical succinctness bound is seldom achieved. We are interested in the 'average succinctness behaviour' of description mechanisms for the regular languages. That is, the relative frequency with which a succinct description is obtained, using a given description mechanism. And, are some mechanisms better than others, in this regard? For example, if the number of states is the criterium, would it be better to use NFAs or statecharts as the description mechanism in a certain application domain?

The theoretical analysis of new description mechanisms is often complex and time-consuming; we needed a practical experiment environment to give a rough indication of the behaviour of new description mechanisms before a theoretical analysis is undertaken. To this end we implemented a **M**odelling **E**nvironment for **R**egular **L**anguages (MERLin).

This paper describes the application of the MERLin system to selective nondeterministic NFAs(⋆-NFAs) [14,15]. In Sect. 2 we define ⋆-NFAs. In Sect. 3 we give a short overview of the MERLin system. We discuss the results obtained from the analysis of certain ⋆-NFAs in MERLin in Sect. 4, and show how this leads to the use of these ⋆-NFAs as random number generators in MERLin.

* This research was supported by grants from the University of Stellenbosch.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 318–326, 2001.

2 Definition of ⋆-NFAs

Van der Walt and Van Zijl introduced ⋆-NFAs in [14]. A detailed analysis of the succinctness achievable by these machines over the regular languages was given in [15]. We recap the main definitions on ⋆-NFAs here.

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

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

$$\delta(A, a) = \overset{\star}{_{q \in A}}\ \delta(q, a) \tag{1}$$

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

δ can also be extended to $\delta : 2^Q \times \Sigma^* \to 2^Q$ in the usual way.

The ⋆-NFA accepts a word w if $\delta(q_0, w)$ contains at least one final state $q_f \in F$.

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

Proof. By the well-known subset construction [6], but use Equation 1 to calculate the transition table of the DFA. See [15] for more details. □

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

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

with δ given by

δ	a
q_1	$\{q_1, q_2\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$.

Choose $\star$ to be union, so that M is a traditional NFA. Use the subset construction to find the DFA $M' = \{Q', \{a\}, \delta', [q_1], F'\}$ equivalent to M. Here δ' is given by:

δ'	a
$[q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_1, q_2, q_3]$
$[q_1, q_2, q_3]$	$[q_1, q_2, q_3]$.

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

δ''	a
$[q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_1, q_3]$
$[q_1, q_3]$	$[q_1, q_2, q_3]$
$[q_1, q_2, q_3]$	$[q_1]$.

□

3 The MERLin System

The MERLin system provides an environment to conduct experiments with automata over the regular languages. It consists of a graphical user interface front-end, the experiment environment itself, and an automata manipulation engine as a back-end.

We adapted existing software to plug in as front-end and back-end. We currently use Grail [13] as our back-end. We ported Grail to Linux, added a $\star$-NFA class, and run one Grail engine on each node of a Beowulf cluster [1]. This allows for parallel processing power in the case of large experiments. MERLin is not dependent on a Beowulf; the user may activate the use of the cluster via menu options if a Beowulf is available. Note that the parallelization is simply a workload distribution over various nodes in the cluster.

The graphical front-end was built using LEDA [9], and the LEDA libraries for graphs and graph manipulation. The graphical user interface contains standard features such as: A finite automata editor; layout algorithms for automata; file storage of automata; and menu options to set up experiments on finite automata.

The MERLin experiment environment is based on finite machines as objects (as in Grail). It allows the user to create one or more finite machines by either (a) reading a pre-defined automaton from a file, or (b) creating an automaton interactively with the automata editor, or (c) using the built-in option to randomly create a number of automata. The resultant automata are stored in files. An experiment is then set up as as a specification (in a menu system) of a series of Grail function calls to be applied to those files. The results are again stored in files which can be accessed through the user interface. The experiment environment also acts as a control shell, for example in controlling the workload distribution of an experiment if the Beowulf option applies.

4 The MERLin Experiment

We set up a MERLin experiment to compare the 'average succinctness behaviour' of different $\star$-NFAs. The results lead to a focused theoretical analysis of, specifically, the unary $\oplus$-NFAs [15]. Here $\oplus$ is used in its usual set-theoretic sense as

$A \oplus B = (A \cup B) \backslash (A \cap B)$. A unary $\star$-NFA is a $\star$-NFA with one alphabet symbol only.

We used MERLin to randomly generate a large number of unary n-state $\star$-NFAs, for $\star$ taken as (respectively) union, intersection and symmetric difference. The same set of $\star$-NFAs was generated in each case by fixing the seed of the random number generator. The final state set was restricted to one state only (state $n-1$), and the start state fixed as state 0. For each set of random $\star$-NFAs, we

- converted the $\star$-NFAs to DFAs,
- minimized the DFAs,
- and retained only unique regular languages.

We analysed the results, and Fig. 1 illustrate typical results, in this case for $n = 5$ states. The figure shows the number of states of the minimal DFAs on the x-axis versus the number of different regular languages accepted by DFAs with this state count on the y-axis. It is noticeable that

- The $\oplus$-NFAs reach the bound $2^n - 1$, whereas the $\cup$-NFAs and $\cap$-NFAs do not (it is known that unary $\cup$-NFAs have the bound $e^{\sqrt{n \log n}}$ [2]).
- There are (relatively) *many* languages that can be represented succinctly with the $\oplus$-NFAs.
- The graph of the $\oplus$-NFAs has many 'gaps', where there are no DFAs with a certain number of states (e.g. 21 to 30).

The $\cap$-NFAs behave similar to $\cup$-NFAs, but the $\oplus$-NFAs show rather remarkable results. We repeated the experiments with different values of n, with different random number generators and different seeds, and with different start and final state sets. The overall results stayed the same. Further theoretical analysis of the unary $\oplus$-NFAs was undertaken.

We define the state cycle of a unary $\oplus$-NFA M as the cycle in its equivalent unary DFA M'. [1] The length of the state cycle of M is the length of the cycle of M'.

Careful scrutiny of the graph in Fig. 1 (and similar graphs for other values of n) reveals that, at each value i which is a factor of $2^n - 1$, there is a relatively large number of DFAs with that cycle length. This fact, together with the 'gaps' mentioned above, is reminiscent of well known results from the theory of switching circuits and the theory of finite fields [3,4].

We showed that a unary $\oplus$-NFA can be encoded so that its state cycle can be seen to be that of a linear feedback shift register (LFSR) over the Galois field GF(2) [15].

[1] In the graphical representation of a unary DFA it is easy to see that every node has a single successor, and the graph hence forms a sequence of nodes. The successor of the last node in this sequence of nodes may be the last node itself, or any of the previous nodes. The successor of the last node therefore determines a cycle in the graph. This cycle can be of length 1 (if the last node returns to itself), or of length k if it returns to the $(k-1)$-th predecessor node of the last node.

Let $\mathcal{F}_i$ denote the i-dimensional vector space of column vectors over GF(2). A linear machine over GF(2) is a 5-tuple

$$M = (\mathcal{F}_k, \mathcal{F}_l, \mathcal{F}_m, \tau, \omega),$$

where $\mathcal{F}_k$ is the set of states, $\mathcal{F}_l$ the set of inputs, $\mathcal{F}_m$ the set of outputs, and τ and ω are linear transformations such that $\tau : \mathcal{F}_{k+l} \to \mathcal{F}_k$ and $\omega : \mathcal{F}_{k+l} \to \mathcal{F}_m$.

The next state $\mathbf{Y}(t)$ of a linear machine at time t can be described as a function of the present state $\mathbf{y}(t)$ and the inputs $\mathbf{x}(t)$. Similarly, the output $\mathbf{z}(t)$ at time t is a function of the present state $\mathbf{y}(t)$ and the inputs $\mathbf{x}(t)$. In matrix notation,

$$\mathbf{Y} = \mathbf{A}\mathbf{y} + \mathbf{B}\mathbf{x} \tag{2}$$

$$\mathbf{z} = \mathbf{C}\mathbf{y} + \mathbf{D}\mathbf{x}. \tag{3}$$

A, B, C and **D** are the characterizing matrices of M.

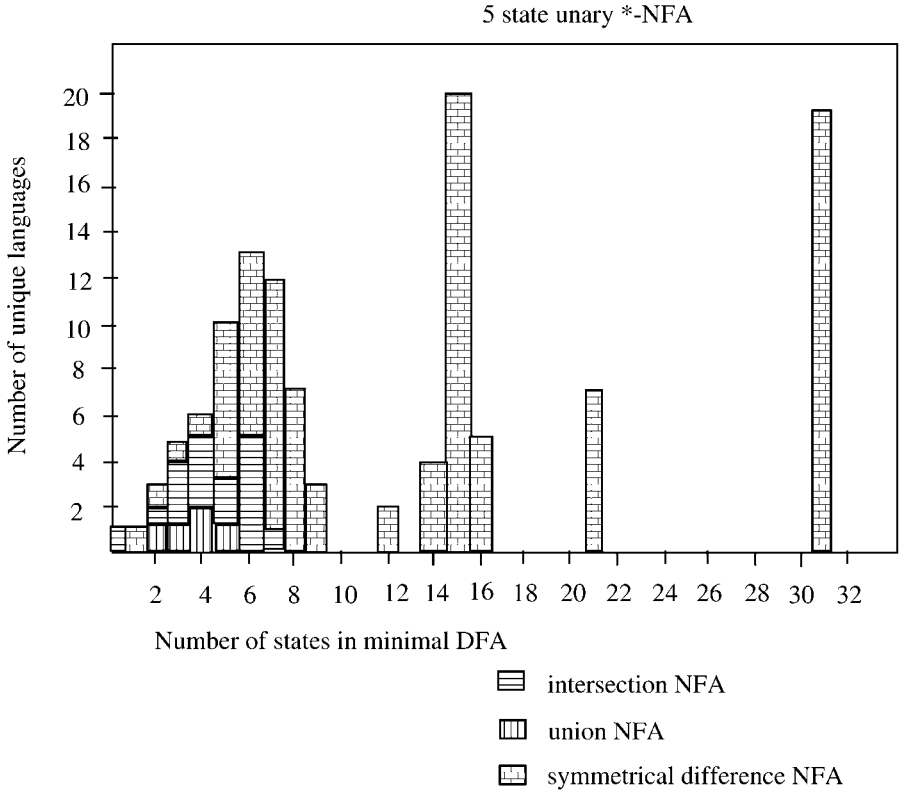


Fig. 1. DFA states vs unique regular languages for 5-state ⋆-NFAs

An *autonomous* linear machine is a linear machine with no input. That is, **B=D=0**, so that the matrix equations (2) and (3) become

$$\mathbf{y}(t) = \mathbf{A}^t\mathbf{y}(0) \tag{4}$$
$$\mathbf{z}(t) = \mathbf{C}\mathbf{y}(t). \tag{5}$$

An LFSR is a linear autonomous machine in which the characterizing matrix $\mathbf{A}$ is an $n \times n$ matrix with the special form

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & \dots & 0 & a_0 \\ 1 & 0 & \dots & 0 & a_1 \\ 0 & 1 & \dots & 0 & a_2 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & a_{n-1} \end{bmatrix},$$

and the characterizing matrix $\mathbf{C}$ is a $1 \times n$ matrix.

The *characteristic polynomial* $c(X)$ of the matrix $\mathbf{A}$ above is given by $\det(X\mathbf{I} - \mathbf{A})$. That is,

$$c(X) = X^n - a_{n-1}X^{n-1} - \ \dots\ - a_1X - a_0.$$

$\mathbf{A}$ is called the *companion matrix* of $c(X)$.

The successive powers of the matrix $\mathbf{A}$ represents the states of the LFSR:

Definition 2. *Let S be an LFSR with characteristic matrix $\mathbf{A}$. Then $\mathbf{A}^k$ represent the states of the LFSR, with $k = 1, 2, \dots, p$ for some integer $p \geq 1$, and p the maximum value for which all the $\mathbf{A}^k$ are distinct.*

Definition 3. *Let S be an LFSR with characteristic matrix $\mathbf{A}$. Let $\mathbf{A}^k$ be the states of the LFSR, with $k = 1, 2, \dots, p$ for some integer $p \geq 1$, and p the maximum value for which all the $\mathbf{A}^k$ are distinct. Then $\mathbf{A}^{k+1} = \mathbf{A}^i$, for some i with $1 \leq i \leq p$. The sequence of states $\mathbf{A}^i, \dots, \mathbf{A}^p$ is the state cycle of the LFSR S.*

Now encode the transition table of a unary $\oplus$-NFA $M = (Q, \Sigma, \delta, q_0, F, \oplus)$ as an $n \times n$ matrix $\mathbf{A} = [a_{ij}]_{n\times n}$ over GF(2); for every state $q_i \in Q$, let

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

It is easy to show by induction that the i-th column of $\mathbf{A}^k$ represents the states reached by M after reading the input word a^k.

Theorem 2. *For any n-state LFSR S, there is a unary $n+1$-state $\oplus$-NFA M with the same state cycle as S.*

Proof. Take any LFSR S with next-state function $\mathbf{y}$. Construct a unary $\oplus$-NFA $M = (Q, \{a\}, \delta, q_0, F)$: Let $Q = \{q_0, q_1, \ldots, q_n\}$ where $q_1, \ldots, q_n$ are the states of S, and q_0 is an additional start state. For every $y_i = 1$ in $\mathbf{y}(0)$, let $q_i \in \delta(q_0, a)$. For $\delta(q_i, a)$, $i > 0$, take the characterizing matrix $\mathbf{A}$ of S to represent the encoding of the transition function of M. Then the j-th state in the state cycle of S is given by $\mathbf{A}^j\mathbf{y}(0)$, which is exactly the j-th state in the DFA equivalent to M. □

A theoretical analysis of unary $\oplus$-NFAs, based on the theorem above, enables one to prove various results about the unary $\oplus$-NFAs. For example,

Theorem 3. *Unary $\oplus$-NFAs are exponentially more succinct than DFAs.*

Proof. Outline (see [15] for more detail): For given n, select any primitive polynomial $c(X)$ of degree n over GF(2). Construct the companion matrix $\mathbf{A}$ of $c(X)$, and let $\mathbf{A}$ be the binary encoding of the transition table of a unary $\oplus$-NFA M_n. Then M_n has maximal period, and the DFA M' equivalent to M_n has size $2^n - 1$. □

Theorem 4. *There exists a family of languages $\{\mathcal{L}_n\}_{n>0}$ such that $\mathcal{L}_n$ is recognized by an n-state $\cup$-NFA M which, when interpreted as a $\oplus$-NFA, also recognizes $\mathcal{L}_n$. Moreover, the smallest DFA recognizing $\mathcal{L}_n$ has $O(2^n)$ states.*

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

$$\delta(i, a) = \{\, (i + (n-1)) \bmod n \quad , i = 0, 1, \ldots, n-1$$

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

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

For any subset A of $\{0, \ldots, n-1\}$, $\bigcap j \in A\delta(A, \sigma) = \emptyset$ 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 [10] proved succinctness. Since the DFAs are identical the result also holds for the $\oplus$-NFA. □

Theorem 5. *There exists a family of languages $\{\mathcal{L}_n\}_{n\geq 1}$ such that $\mathcal{L}_n$ is recognized by an n-state $\cap$-NFA M which, when interpreted as a $\oplus$-NFA, also recognizes $\mathcal{L}_n$.*

Proof. See [15]. □

4.1 Random Generation of Finite Automata

It is known that LFSRs can be used as random number generators [7]. The results of the previous section allow us to use unary ⊕-NFAs as the random number generators in MERLin.

To generate random numbers with unary ⊕-NFAs, we apply the following six-step process:

1. Encode the generator unary ⊕-NFA M into a matrix $\mathbf{A}$ as described previously.
2. Convert M to its equivalent DFA step by step, that is, compute $\mathbf{A}^k$, for $k = 0, \ldots, p$, where p is the cycle length of M.
3. For each $\mathbf{A}^k$, compute $\mathbf{A}^k\mathbf{y}(0)$. Here $\mathbf{y}(0)$ is the seed for the random sequence, which is formed by encoding the start states for M into an $n \times 1$ column vector.
4. The sequence $\mathbf{y}(0)$,$\mathbf{A}\mathbf{y}(0)$,$\mathbf{A}^2\mathbf{y}(0)$, $\mathbf{A}^3\mathbf{y}(0), \ldots$ calculated above is a sequence of $n \times 1$ vectors. Take from each vector the element y_1; these elements form a sequence of bits.
5. Take the bit sequence obtained above, and group it into equal-sized groups groups of bits.
6. Take each group to represent the binary representation of a whole number. This sequence of numbers forms the pseudo-random sequence.

By the correspondence between LFSRs and unary ⊕-NFAs, it is trivial to show that the unary ⊕-NFAs perform as well as LFSRs as random number generators. It was shown for LFSRs [8] that characteristic polynomials of the form $q^k - q^s - 1$ must be combined to obtain pseudorandom sequences with good statistical properties. Such combined generators are formed by applying the six-step process above to (usually three) different unary ⊕-NFAs, and taking the symmetric difference of the different bit sequences after the fourth step.

To randomly generate an n-state ⋆-NFA with k alphabet symbols, we group the bit stream into blocks of size kn^2. This block is interpreted as the transition table of a ⋆-NFA with k alphabet symbols. The set of start states and set of final states are generated by two other independent streams, each with block size n.

The random generation of an n-state ⋆-NFA includes *disconnected* n-state ⋆-NFAs. Previous work [11] overcame this problem by manually connecting a generated ⋆-NFA; another approach may be to discard disconnected ⋆-NFAs. Both these approaches change the original pseudo-random bitstream, and therefore compromise the integrity of the random objects. In MERLin we overcome the problem by interpreting the results of experiments on randomly generated n-state ⋆-NFAs as results on ⋆-NFAs with n or less states (that is, connected and disconnected n-state ⋆-NFAs).

Our method of randomly generating ⋆-NFAs are based on a mapping from the transition tables of the ⋆-NFAs to numbers – the block of size kn^2 represents one number in the pseudo-random sequence. However, this does not guarantee in any way that the stream of generated ⋆-NFAs are random over the domain of the regular languages. We are currently investigating methods to map the

pseudo-random $\star$-NFA stream to an enumeration of the regular languages in order to test the quality of the pseudo-random $\star$-NFA stream over this domain.

5 Conclusion and Future Work

We described a Modelling Environment for Regular Languages (MERLin), which can be used to conduct experiments on finite automata and regular languages. We illustrated the use of the MERLin system in comparing the typical descriptional complexity behaviour of different types of finite machines.

We are currently extending the MERLin software, with: A random number generator test suite; a cellular automaton class in Grail; and a visual display component for automaton execution trees.

References

1. T.L. Stirling et al, How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters. MIT Press, Scientific and Engineering Computation Series, 1999.
2. M. Chrobak, Finite Automata and Unary Languages. Theoretical Computer Science **47**(1986), pp. 149–158.
3. L.L. Dornhoff and F.E. Hohn, Applied Modern Algebra. MacMillan Publishing Co., Inc., New York, 1977.
4. S.W. Golomb, Shift Register Sequences. Holden-Day, Inc., 1967.
5. D. Harel, Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8**(1987), pp. 231–274.
6. J.E. Hopcroft and J.D. Ullman, Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Massachusetts, 1979.
7. P. L'Ecuyer, Maximally Equidistributed Combined Tausworthe Generators. Mathematics of Computation **65**(213):203–213, 1996.
8. P. L'Ecuyer, Testing Random Number Generators. Proceedings of the 1992 Winter Simulation Conference, IEEE Press, Dec. 1992, pp.305–313.
9. K. Mehlhorn *et al*, The LEDA User Manual Version 4.0. Max-Planck-Institut für Informatik, Saarbrücken, Germany. *http://www.mpi-sb.mpg.de/LEDA*
10. E. Leiss, Succinct Representation of Regular Languages by Boolean Automata. Theoretical Computer Science **13**(1981), pp. 323–330.
11. T.K.S. Leslie, Efficient Approaches to Subset Construction. MSc Thesis, University of Waterloo, Waterloo, Canada, 1994.
12. A.R. Meyer and M.J. Fischer, Economy of Description by Automata, Grammars, and Formal Systems. Proc. of the 12th Annual IEEE Symposium on Switching and Automata Theory, October 1971, Michigan, pp. 188-191.
13. D.Raymond and D. Wood, The User's Guide to Grail. University of Waterloo, Waterloo, Canada, 1995. *http://www.csd.uwo.ca/research/grail*
14. A.P.J. van der Walt and L. van Zijl, $\star$-Realizations of Deterministic Finite Automata. British Colloquium for Theoretical Computer Science 11, Swansea, Wallis, April 1995.
15. L. van Zijl, Generalized Nondeterminism and the Succinct Representation of Regular Languages. PhD dissertation, Stellenbosch University, March 1997. *http://www.cs.sun.ac.za/~lynette/boek.ps.gz*

Visual Exploration of Generation Algorithms for Finite Automata on the Web

Stephan Diehl, Andreas Kerren, and Torsten Weller

University of Saarland, FR 6.2 Informatik,
PO Box 15 11 50, D-66041 Saarbrücken, Germany
{diehl, kerren}@cs.uni-sb.de

1 Introduction

In previous articles we presented a PC based educational software on lexical analysis [1] and semantical analysis [4]. These systems were developed using an authoring system under MS Windows. The user can not change regular expressions or input words for nondeterministic respectively deterministic finite automata. To overcome these restrictions we developed GaniFA, our Java applet for visualization of algorithms from automata theory. It can be downloaded from our web page `http://www.cs.uni-sb.de/GANIMAL` as a JAR-file and requires a Java Plug-In 1.2. We invite the reader to use it in own web-based exercises, lecture notes or presentations on finite automata. Furthermore this web page gives a short overview, how this applet can be customized and embedded into HTML web pages.

2 GaniFA

The GaniFA applet visualizes and animates the following algorithms [5]:

- Generation of a non-deterministic finite automaton (NFA) from a regular expression RE, see Figure 1.
- Removal of ε-transitions of a NFA.
- Transformation of a deterministic finite automaton (DFA) from a NFA without ε-transitions.
- Minimization of a deterministic finite automaton (minDFA).
- For each of the above automata generated above, the applet can visualize the computation of the automaton on an input word.

GaniFA is customizable through a large set of parameters. In particular, it is possible to visualize only some of the algorithms and to pass a finite automaton or a regular expression as well as an input word to the applet. The GaniFA applet was embedded into an electronic textbook on the theory of finite automata, which can be studied with the help of a usual web browser like Netscape Communicator or MS Internet Explorer. Currently there are English and German versions of the textbook and of the applet itself.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 327–328, 2001.

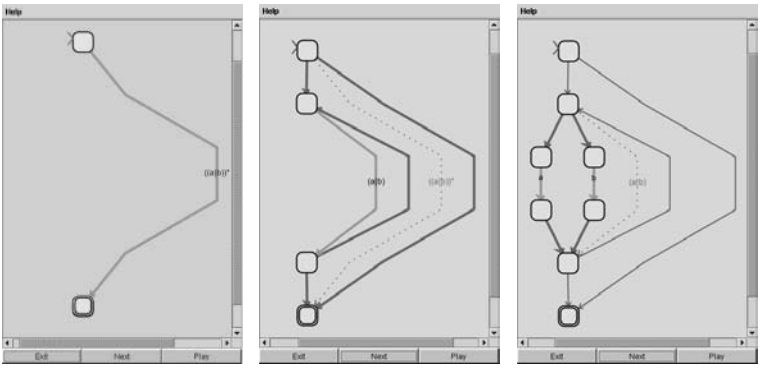


Fig. 1. Layout of the intermediate and final NFA for the RE $(a|b)^*$.

3 Conclusion

Although GaniFA and our electronic textbook only cover a small part of the theory on generating finite automata, they can be very useful for introductory courses. They provide a new way to access the material and allow for explorative, self-controlled learning [3]. Teachers can not only use our textbook as it is, but they can also embed GaniFA in their on lecture notes and exercises. As part of our future work, we plan to use the technical framework underlying GaniFA and GANIMAM [2] to implement customizable, interactive web-based visualizations of other computational models[1].

References

1. B. Braune, S. Diehl, A. Kerren, R. Wilhelm. *Animation of the Generation and Computation of Finite Automata for Learning Software.* To appear in Proceedings of Workshop of Implementing Automata WIA'99, Potsdam, Germany, July, 1999.
2. S. Diehl, T. Kunze. *Visualizing Principles of Abstract Machines by Generating Interactive Animations.* In Future Generation Computer Systems, Vol. 16(7), Elsevier, 2000.
3. S. Diehl, A. Kerren. *Increasing Explorativity by Generation.* In Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications, EDMEDIA-2000, AACE, Montreal, Canada, 2000.
4. A. Kerren. *Animation of the Semantical Analysis.* In Proceedings of "8. GI-Fachtagung Informatik und Schule INFOS99" (in German), pp. 108-120, Informatik aktuell, Springer, 1999.
5. R. Wilhelm, D. Maurer. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 1995.

[1] This research has been partially supported by the German Research Council (DFG) under grant WI 576/8-1 and WI 576/8-3.

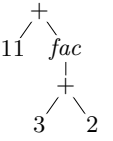
TREEBAG*

Frank Drewes[1] and Renate Klempien-Hinrichs[2]

[1] Department of Computing Science, Umeå University
S-901 87 Umeå (Sweden)
drewes@cs.umu.se
[2] Department of Computer Science, University of Bremen
P.O.Box 33 04 40, D-28334 Bremen (Germany)
rena@informatik.uni-bremen.de

TREEBAG is a system that allows to generate and transform objects of several types. This is accomplished by generating and transforming trees using *tree grammars* and *tree transducers*, and interpreting the resulting trees as expressions that denote objects of the desired type. For this, *algebras* are used, each algebra defining an abstract data type (i.e., a set of objects together with a number of operations on them). Finally, there are *displays* whose purpose is to visualise the generated objects. Tree grammars, tree transducers, algebras, and displays are the four types of *TREEBAG components* everything relies on.

In order to explain how TREEBAG works, some basic concepts are required. By a *tree* we mean a rooted and ordered tree with node labels taken from a ranked alphabet (or *signature*). A node which is labelled with a symbol of rank n must have exactly n children. For example, the signature $\Sigma = \{+\colon 2, fac\colon 1\} \cup \{c\colon 0 \mid c \in \mathbb{N}\}$ contains the symbols + and *fac* of rank 2 and 1, respectively, and all natural numbers, each considered as a symbol of rank 0. One of the trees over this signature is shown on the right. Seen as a term in the natural way, it would be denoted $+[11, fac[+[3, 2]]]$ or, using infix notation for the binary symbol + in order to enhance readability, $11 + fac[3 + 2]$.



An algebra interprets every symbol of a signature as an operation on the domain of that algebra (where arities of operations and ranks of symbols coincide). Thus, every tree may be considered as an expression that denotes an element of the domain. Taking the signature Σ from above as an example, one may, choose the domain $\mathbb{N}$ and interpret + as addition, *fac* as the faculty function, and $c \in \mathbb{N}$ as c. Then the tree depicted above denotes the number 131. It is important to notice that the signatures and trees themselves do not have a particular meaning—one can as well consider a totally different algebra to interpret the symbols in Σ. A tree as such is pure syntax without meaning.

A tree grammar is any device that generates a language of trees. A tree transducer transforms input trees into output trees according to some (possibly

* This research was partially supported by the EC TMR Network GETGRATS, the ESPRIT Basic Research Working Group APPLIGRAPH, and the *Deutsche Forschungsgesellschaft (DFG)* under grant no. Kr-964/6-1

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 329–330, 2001.

complex) rule. Now, if one interprets the output trees of a tree grammar or tree transducer by means of an appropriate algebra, objects from the respective domain are obtained. Thus, one can deal with all kinds of objects in a tree-oriented way just by choosing a suitable algebra to interpret the trees. Many of the well-known systems studied in formal language theory can be simulated nicely in this way. For a more detailed and formal discussion see, e.g., [DE98, Dre98a,Dre98b].

TREEBAG allows to arrange instances of the four types of components as nodes of an acyclic graph and to establish input/output relations between them. More precisely, the output of a tree grammar or tree transducer can be fed into tree transducers and displays. Furthermore, in order to make a display work, an algebra must be associated with it. Then the display will interpret its input trees accordingly and visualise the resulting objects.

The system is implemented in pure Java, and it must be stressed that each of the mentioned types of TREEBAG components consists of several classes. There is, for example, one class that implements top-down tree transducers and another one that implements the so-called YIELD transduction. Both are tree transductions, but of a different type. In fact, due to modularity one can easily add new types of tree transducers, simply by implementing a corresponding class.

Every class of TREEBAG components defines its own syntax. This makes it possible to load an instance of the class—a concrete regular tree grammar, for example—from a file. Furthermore, every class provides a set of commands for interaction. So far, the following TREEBAG components are available:

- regular, ETOL, and parallel deterministic total tree grammars;
- top-down tree transducers, the YIELD transduction, and a meta-class of tree transducers called iterator;
- algebras on truth values, integers, strings, trees, and two-dimensional collages, as well as algebras that correspond to the chain-code and turtle formalisms, yielding line drawings;
- displays for a textual representation of objects (which can be used to display truth values, numbers, strings, and trees), for a graphical representation of trees, and for collages and line drawings.

As arbitrary compositions of tree grammars and tree transducers can be built, the availability of these basic classes opens up a large number of possibilities.

The newest version of TREEBAG, including examples ranging from a prime test on natural numbers to the generation of celtic knotwork, is available at `http://www.informatik.uni-bremen.de/~drewes/treebag` .

References

[DE98] Frank Drewes and Joost Engelfriet. Decidability of the finiteness of ranges of tree transductions. *Information and Computation*, 145:1–50, 1998.

[Dre98a] Frank Drewes. TREEBAG—a tree-based generator for objects of various types. Report 1/98, Univ. Bremen, 1998.

[Dre98b] Frank Drewes. Tree-based picture generation. Report 7/98, Univ. Bremen, 1998. Revised version to appear in *Theoretical Computer Science*.

Word Random Access Compression*

Jiří Dvorský and Václav Snášel

Computer Science Department, Palacky University of Olomouc, Tomkova 40,
779 00 Olomouc, Czech Republic
{jiri.dvorsky,vaclav.snasel}@upol.cz

Abstract. Compression method (WRAC) based on finite automatons is presented in this paper. Simple algorithm for construction finite automaton for given regular expression is shown. The best advantage of this algorithm is the possibility of random access to a compressed text. The compression ratio achieved is fairly good. The method is independent on source alphabet i.e. algorithm can be character or word based.

Keywords. word-based compression, text databases, information retrieval, HuffWord, WLZW

1 Random Access Compression

Let be $A = \{a_1, a_2, \ldots, a_n\}$ an alphabet. Document D of length m can be written as sequence $D = d_0, d_1, \ldots, d_{m-1}$, where $d_i \in A$. For each position i we are able to find out which symbol d_i is at this position. We must save this property to create compressed document with random access.

A set of position $\{i; 0 \leq i < m\}$ can be written as a set of binary words $\{b_i\}$ of fixed length. This set can be considered as language $L(D)$ on alphabet $\{0, 1\}$. It can be easy shown that the language $L(D)$ is regular ($L(D)$ is finite) and it is possible to construct DFA which accepts the language $L(D)$. This DFA can be created, for example, by algorithm given in [3]. Regular expression is formed as $b_0 + b_1 + \cdots + b_{m-1}$.

Compression of the document D consists in creating a corresponding DFA. But decompression is impossible. The DFA for the document D can only decide, whether binary word b_i belongs to the language $L(D)$ or not. The DFA does not say anything about a symbol which appears in position i. Inorder to do this, the definition of DFA must be extended.

Definition 1. *A deterministic finite automaton with output (DFAO) is a 7-tuple $(Q, A, B, \delta, \sigma, q_0, F)$, where Q is a finite set of states, A is a finite set of input symbols (input alphabet), B is a finite set of output symbols (output alphabet), δ is a state transition function $Q \times A \to Q$, q_0 is the initial state, σ is an output function $F \to B$, $F \subseteq Q$ is the set of final states.*

* This work was done under grant from the Grant Agency of Czech Republic, Prague No.: 201/00/1031

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 331–332, 2001.

This type of automaton is able to determine for each of the accepted words b_i which symbol lies on position i. To create an automaton of such a type the algorithm mentioned in [3] must be extended too. Regular expression V, which is input into the algorithm, consists of words b_i. Each b_i must carry its output symbol d_i. Regular expression is now formed as $b_0d_0 + b_1d_1 + \cdots + b_{m-1}d_{m-1}$,

The set of states Q of the automaton $DFAO(V)$ is divided into disjunct subsets (so called *layers*). Transitions are done only between two adjacent layers. Thus states can be numbered locally in those layer. Final automaton is stored on disk after construction. All layers are stored sequentially.

Let's remark, that algorithm of construction of automaton is independent with respect to its output alphabet. There are two possibilities. The first is a classic character based version. Algorithm is one-pass and output alphabet is a standard ASCII. For the text retrieval systems word-based version (the second possibility) is more advantageous because of the character of natural languages.

2 Experimental Results

To allow practical comparison of algorithm, experiments have been performed on some compression corpus. For test has been used Canterbury Compression Corpus (large files), especially King's James Bible (bible.txt) file which is $4,077,774$ bytes long. There are $153,5710$ tokens and $13,461$ of them are distinct. A word-based version of algorithm has been used for a test.

Implementation of this method is described in [2].

Table 1. Comparison with other compression utilities

Compression utility	Compressed text [bytes]	Ratio [%]
WRAC	1480884	36.3
ARJ 2.41a	1207114	29.6
WINZIP 6.2	1178869	28.9
GZip (UNIX)	1178757	28.9
WinRAR 2.0	994346	24.4
WLZW (word-based LZW)	896956	22.0

References

1. J. Dvorský, V. Snášel, J. Pokorný. *Word-based Compression Methods for Large Text Documents.* Data Compression Conferencs - DCC '99, Snowbird, Utah USA.
2. J. Dvorský. *Text Compression with Random Access.* Workshop ISM 2000, Czech Republic, ISBN 80-85988-45-3
3. G. Rozenberg, A.Salomaa, Ed. *Handbook of Formal Language.* Springer Verlag 1997, Vol. I.–III.

Extended Sequentialization of Transducers

Tamás Gaál

Xerox Research Centre Europe - Grenoble Laboratory
6, chemin de Maupertuis, 38240 Meylan, France

tamas.gaal@xrce.xerox.com http://www.xrce.xerox.com

Sequential transducers, introduced by Schützenberger [5], have advantageous computational properties. A sequential transducer is deterministic with respect to its input. Not all transducers can be sequentialized: but if one can be, it means time, and, often, space optimality. This article extends the subsequentialization algorithm of Mohri [3,4] for previously untreated classes of transducers. We

- change the representation of *final p-strings*,
- extend the sequentialization to input ε labels and their closures,
- handle the *unknown symbol*.

Mohri uses *final p-strings* to express p-subsequentiality. We convert them to real arcs and states to have a more uniform representation and to maintain the two-sided applicability of the transducer. This change is of linear complexity.

An ε-closure set and appropriate modifications in the subsequentialization algorithm of Mohri make it possible to handle transducers containing input-side ε labels. This does not require any intermediate transformation of the transducer.

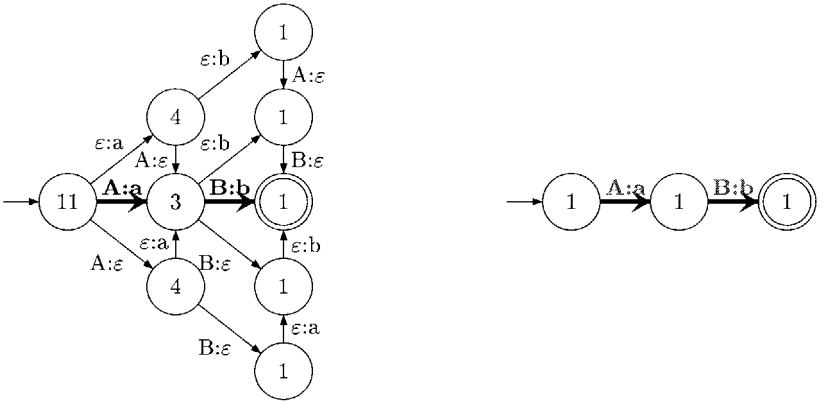


Fig. 1. The left-hand transducer has all possible ambiguities of transducing a string of 2 symbols to another string of 2 symbols (`AB` to/from `ab`). Its equivalent, by extended ε-closure sequentialization, is a linear one (on the right). The left network has 11 paths; 10 of them are spurious. The states of the networks are annotated with the number of paths leading from the given state to a final state.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 333–334, 2001.

Our ε-closure modification solves a complexity problem in subsequentiable transducers that contain arcs with an input ε label either on the input or on the output side. An illustration of such cases is the transduction of a string of length n to another string of the same size (Fig. 1). Such a transducer can be ambiguous according to a rapidly growing function in n; a (modest) lower bound for the number of possible paths is $\mathcal{O}(\binom{2n}{n})$, for which a lower bound is $\mathcal{O}(3^n)$, that is, such a transducer has an exponential number of ambiguous paths expressing the same mapping. If such a transducer is Kleene star-red (allowing repetitions of the input string), k repetitions will require more than $\mathcal{O}(\binom{2n}{n}^k)$ recognition complexity. We only know a recursive form for the number of possible paths in the general case but lower bound approximations show that such cases become rapidly untreatable. But such transducers can be transformed, by ε-sequentialization, making recognition complexity linear, $\mathcal{O}(nk)$.

By using the ε-closure, the ambiguities stemming from input ε transitions can be handled, and both ordinary and ε-ambiguities are (sub)sequentialized in the same step, by local modifications. This extension does not increase the complexity of the original algorithm. The ε-closure operation is a semiring operation creating a set of directed acyclic graphs. Such ε-ambiguities may and do arise in finite-state compilers and tools.

The *unknown symbol* is an extension of the usual transducer notation to define special treatment for input symbols not in the input alphabet of the transducer [1]. It can be present, if handled specially, in sequentialization and in subsequent finite-state calculus operations and applications. The solution is local at sequentialization time, with no additional complexity, and needs a run-time queue of bounded size; this solution has been reused in another finite-state implementation [6].

The beneficial effects of these transformations have been used in real natural language processing cases. They confirm the practical results of Mohri: 14-40% typical efficiency improvement was measured after subsequentializing lexical transducers created by Xerox finite-state tools [2].

References

1. T. Gaál and L. Karttunen. Improving Mohri's algorithm in the Xerox finite state calculus. In *Proceedings of the 9th International Conference on Automata and Formal Languages (AFL'99)*, 1999. To appear in Publicationes Mathematicae.
2. L. Karttunen and K.R. Beesley. *Finite-State Morphology: Xerox Tools and Techniques*. Cambridge University Press, Cambridge UK, 2000? Forthcoming.
3. M. Mohri. Compact representation by finite-state transducers. In *Proceedings of the 32nd meeting of the Association for Computational Linguistics (ACL 94)*, 1994.
4. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, pages 269–312, 1997.
5. M.P. Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, 1977.
6. G. van Noord and D. Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Proceedings of the Workshop on Implementing Automata (WIA'99)*, LNCS, Potsdam, 1999. Springer. To appear.

Lessons from INR in the Specification of Transductions

J. Howard Johnson

Institute for Information Technology, National Research Council Canada
Howard.Johnson@nrc.ca

Abstract. Finite state transductions have been shown to be quite useful in a number of areas; however, it is still the case that it is often difficult to express certain kinds of transductions without resorting to a state and transition view. INR was developed to explore this problem, and several applications of transduction were studied as exercises in specification during INR's development. The specification of the NYSIIS phonetic encoding function (developed for the New York State Identification and Intelligence System) provides a clear example of many important ideas. An INR specification for NYSIIS is provided, which is syntactically similar to the prose description and from which INR can directly produce the 149 state subsequential transducer.

Phonetic encoding of names has been used to support search in large databases based on surnames which may have been misspelled. The best known of these is Soundex, but it has been found to have many defects for common types of names. The NYSIIS encoding function [3,2] was designed as a replacement to improve the handling of Spanish and southern European surnames.

A prose description of NYSIIS has been converted to INR syntax [1] (see Figure 1), resulting in a specification that mirrors the structure of the original description and yet can be directly converted to a subsequential transducer and compiled into a highly optimized subroutine. This facilitates greater flexibility in the design of phonetic encoding functions while preserving implementation efficiency. Adding the capability for representing the features of NYSIIS to INR has introduced many new operators and constructions, described fully in the longer version of this paper [2].

References

1. Johnson, J. H.: INR: A program for computing finite automata. http://www.seg.iit.nrc.ca/~johnson/pubs/INR86.html (1986)
2. Johnson, J. H.: Lessons from INR in the specification of transductions. http://www.seg.iit.nrc.ca/~johnson/pubs/LIST.html (2000)
3. Moore, G. B., Kuhns, J. L., Trefftzs, J. L., Montgomery, C. A.: Accessing individual records from personal data files using non-unique identifiers. Technical Report NBS Special Publication 500–2, U.S. Dept. of Commerce—National Bureau of Standards (1977) Available from the National Technical Information Service

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 335–336, 2001.

```
Vowel     = { A, E, I, O, U };
Consonant = { B, C, D, F, G, H, J, K, L, M, N,
              P, Q, R, S, T, V, W, X, Y, Z };
Letter    = Consonant | Vowel;
Copy1     = Letter $ (0,0);
Copy      = Copy1*;
BeforeEqualsAfter = ( Copy1 '_' Copy1 ) @ ( Letter $ 0 0 );
SetToPreceding = ( Copy1 ( '_', '_' ) ( Letter, Letter ) )
             @@ BeforeEqualsAfter;

Ny1 = ( 'MAC', 'MCC' ) Copy
   || ( 'KN',  'NN' )  Copy
   || ( 'K',   'C' )   Copy
   || ( 'PH',  'FF' )  Copy
   || ( 'PF',  'FF' )  Copy
   || ( 'SCH', 'SSS' ) Copy
   ||                  Copy;
Ny2 = Copy ( 'EE', 'Y' )
   || Copy ( 'IE', 'Y' )
   || Copy ( {'DT','RT','RD','NT','ND'}, 'D' )
   || Copy;
Ny34 = Copy1 ['_'] Copy;
Ny5a = Copy ( '_', ^ )
    || Copy ( '_EV',     '_AF' ) Copy
    || Copy ( '_' Vowel, '_A'  ) Copy;
Ny5b = Copy ( '_Q', '_G' ) Copy
    || Copy ( '_Z', '_S' ) Copy
    || Copy ( '_M', '_N' ) Copy;
Ny5c = Copy ( '_KN', '_NN' ) Copy
    || Copy ( '_K',  '_C'  ) Copy;
Ny5d = Copy ( '_SCH', '_SSS' ) Copy
    || Copy ( '_PH',  '_FF'  ) Copy;
Ny5e = ( Letter* ( Consonant '_H' | '_H' Consonant ) Letter* )
       @@ ( Copy SetToPreceding Copy );
Ny5f = ( Letter* Vowel '_W' Letter* ) @@ ( Copy SetToPreceding Copy );
Ny5 = Ny5a || Ny5b || Ny5c || Ny5d || Ny5e || Ny5f
          || Copy ( '_', '_' ) Copy;
Ny6 = ( ( Letter* BeforeEqualsAfter Letter* )
        @@ ( Copy '_' Letter ['_'] Copy ) )
   || ( Copy '_' Copy1 ['_'] Copy );
Ny3456 = Ny34 @ ( Ny5 @ Ny6 :clsseq );
Ny7 = Copy ( 'S', ^ )    || Copy;
Ny8 = Copy ( 'AY', 'Y' ) || Copy;
Ny9 = Copy ( 'A', ^ )    || Copy;

NYSIIS = Ny1 @ Ny2 @ Ny3456 @ Ny7 @ Ny8 @ Ny9 :sseq;
```

Fig. 1. NYSIIS as an INR specification

Part-of-Speech Tagging with Two Sequential Transducers

André Kempe

Xerox Research Centre Europe – Grenoble Laboratory
6 chemin de Maupertuis – 38240 Meylan – France
andre.kempe@xrce.xerox.com – http://www.xrce.xerox.com/research/mltt

1 Introduction

We present a method of constructing and using a cascade consisting of a left- and a right-sequential finite-state transducer (FST), T_1 and T_2, for part-of-speech (POS) disambiguation. Compared to a Hidden Markov model (HMM), this FST cascade has the advantage of significantly higher processing speed, but at the cost of slightly lower accuracy. Applications such as Information Retrieval, where the speed can be more important than accuracy, could benefit from this approach.

In the process of POS tagging, we first assign every word of a sentence a unique ambiguity class c_i that can be looked up in a lexicon encoded by a sequential FST. Every c_i is denoted by a single symbol, e.g. "`[ADJ NOUN]`", although it represents a set of alternative tags that a given word can occur with. The sequence of the c_i of all words of one sentence is the input to our FST cascade (Fig. 1). It is mapped by T_1, from left to right, to a sequence of reduced ambiguity classes r_i. Every r_i is denoted by a single symbol, although it represents a set of alternative tags. Intuitively, T_1 eliminates the less likely tags from c_i, thus creating r_i. Finally, T_2 maps the sequence of r_i, from right to left, to an output sequence of single POS tags t_i. Intuitively, T_2 selects the most likely t_i from every r_i (Fig. 1).

Although our approach is related to the concept of bimachines [2] and factorization [1], we proceed differently in that we build two sequential FSTs directly and not by factorization.

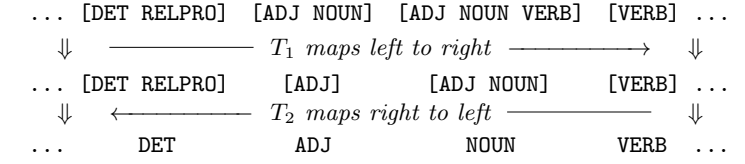


Fig. 1. Input, intermediate, and output sequence

2 Construction of the FSTs

In T_1, one state is created for every r_i (output symbol), and is labeled with this r_i (Fig. 2a). An initial state, not corresponding to any r_i, is created in addition.

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 337–339, 2001.

From every state, one outgoing arc is created for every c_i (input symbol), and is labeled with this c_i. The destination of every arc is the state of the most likely r_i in the context of both the current c_i (arc label) and the preceding r_{i-1} (source state label). This most likely r_i is estimated from the transition and emission probabilities of the different r_i and c_i. Then, all arc labels are changed from simple symbols c_i to symbol pairs $c_i:r_i$ (mapping c_i to r_i) that consist of the original arc label and the destination state label. All state labels are removed (Fig. 2b). Those r_i that are unlikely in any context disappear, after minimization, from T_1. T_1 accepts any sequence of c_i and maps it, from left to right, to the sequence of the most likely r_i in the given left context.

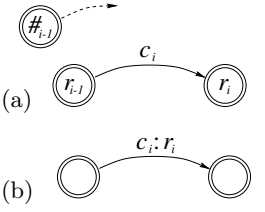


Fig. 2. Construction of T_1

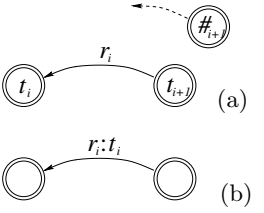


Fig. 3. Construction of T_2

In T_2, one state is created for every t_i (output symbol), and is labeled with this t_i (Fig. 3a). An initial state is added. From every state, one outgoing arc is created for every r_i (input symbol) that occurs in the output language of T_1, and is labeled with this r_i. The destination of every arc is the state of the most likely t_i in the context of both the current r_i (arc label) and the following t_{i+1} (source state label). Note, this is the following tag, rather than the preceding, because T_2 will be applied from right to left. The most likely t_i is estimated from the transition and emission probabilities of the different t_i and r_i. Then, all arc labels are changed into symbol pairs $r_i:t_i$ and all state labels are removed (Fig. 3b), as was done in T_1. T_2 accepts any sequence of r_i generated by T_1 and maps it, from right to left, to the sequence of the most likely t_i in the given right context.

Both T_1 and T_2 are sequential. They can be minimized with standard algorithms. Once T_1 and T_2 are built, the transition and emission probabilities of all t_i, r_i, and c_i are of no further use. Probabilities do not (directly) occur in the FSTs, and are not (directly) used at run time. They are, however, "implicitly contained" in structure of the FSTs.

3 Results

We compared our FST tagger on 3 languages (English, German, Spanish) with a commercially available HMM tagger. The FST tagger was on average 10 times as fast but slightly less accurate than the HMM tagger (45 600 words/sec and 96.97% versus 4 360 words/sec and 97.43%). In some applications such as Information Retrieval a significant speed increase can be worth the small loss in accuracy.

References

1. C. C. Elgot, and J. E. Mezei. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development*, pages 47–68, January.
2. M. P. Schützenberger. 1961. A remark on finite transducers. *Information and Control*, 4:185–187.

Solving Complex Problems Efficiently with Adaptive Automata

João José Neto

Escola Politécnica da Universidade de São Paulo - Dep. de Eng. de Computação e Sistemas Digitais Av. Prof. Luciano Gualberto, trav. 3, n. 158 CEP 05508-900 - Cidade Universitária - São Paulo - SP - Brasil
jjneto@pcs.usp.br

Abstract. Adaptive technologies [**1**] are based on the self-modifying property of some systems, which give their users a powerful facility for expressing and handling complex problems. One may turn a rule-based formalism into a corresponding adaptive one by attaching adaptive actions to their rules. This work focuses adaptive automata, and adaptive formalism based on structured pushdown automata. Its transitions may hold adaptive actions responsible for self-modifications. An example ilustrates an adaptive-automata-based solution to an example problem focusing the copy language, an interesting context-dependent language.

Structured pushdown automata are state machines composed by a set of finite-state-like mutually recursive *sub-machines*. In *adaptive automata* [**2**], [**3**] such rules may be attached *adaptive actions*. *(γg, e, sα), A : → (γg', e', s'α), B* represents the general form of a rule in an adaptive automaton. Its left-hand side refers to the configuration of the automaton before, whereas the right-hand side encodes its configuration after the state transition. The components of the 3-tuples encode the situation of the pushdown store, the state and the input data, respectively. Adaptive actions (A and B) are optionally specified: the left one represents modifications to be applied before the state transition, while the right one specifies the changes to be imposed to the automaton after the transition. Adaptive actions are calls to parametric *adaptive functions* representing collections of *elementary adaptive actions* to be applied to the transition set of the automaton. Three elementary adaptive actions are allowed: inspection, deletion and insertion of transitions. ⊗ *[(γg, e, sα), A : → (γg', e', s'α), B]* denotes any elementary adaptive actions by replacing the operator ⊗ by ***?*** for the inspection, **+** for the insertion and − for the deletion of transitions having the shape specified in brackets. Adaptive automata are Turing-powerful, so they can handle context-sensitive languages.

Example. The following example ilustrates the use of adaptive automata to implement an acceptor for the copy language L=ww ⊢ with w ∈ Σ^*. This is neither a regular nor a context-free language. It is useful in the processing of reduplication, a general linguistic construct present in some far-eastern natural languages, in which some words are formed by concatenating two copies of another word. One adaptive possible acceptor for this language has two submachines:

S. Yu and A. Păun (Eds.): CIAA 2000, LNCS 2088, pp. 340–342, 2001.

- **The main submachine**, responsible for the global structure of the sentence, (1) reads the input string; (2) builds a section of the main submachine that allows reprocessing the input string; (3) detects the end of the input string; (4) transfers control to the extension built in item 2, where: (5) for each transition, the attached symbol is stacked back onto the input string; (6) after restoring the whole input string, the secondary submachine is called twice; (7) the input string is accepted if and only if the final state is reached.
- **An auxiliary submachine** detects the occurrence of duplications within the sentence: (1) re-reads the first half of the input string when called the first time; (2) tries to match the second half of the sentence when called again.

The listing below shows the full formalization of such an adaptive automaton.

the main submachine
mounting both string halves:
$(\gamma, e1, \sigma\alpha) \rightarrow (\gamma, e2, \alpha)$, $A(\sigma)$
$(\gamma, e2, \sigma\alpha) \rightarrow (\gamma, e1, \alpha)$, $B(\sigma)$
prepare to re-read:
$(\gamma, e1, \vdash\alpha) \rightarrow (\gamma, e6, \alpha)$,C ( )
re-consuming both halves:
$(\gamma, e3, \alpha) \rightarrow (\gamma e4, e5, \alpha)$
$(\gamma, e4\ \alpha) \rightarrow (\gamma ef, e5, \alpha)$
final state:
$(\gamma, ef, \alpha) \rightarrow (\gamma, ef, \alpha)$
the auxiliary submachine
(built by the adaptive actions)
returns to calling submachine:
$(\gamma e4, ef1, \alpha) \rightarrow (\gamma, e4, \alpha)$
$(\gamma ef, ef1, \alpha) \rightarrow (\gamma, ef, \alpha)$
auxiliary (pointer)
transitions
$(\gamma, x, \alpha) \rightarrow (\gamma, e5, \alpha)$
$(\gamma, y, \alpha) \rightarrow (\gamma, e6, \alpha)$
$(\gamma, m, \alpha) \rightarrow (\gamma, e5, \alpha)$
adaptive actions
build the auxiliary submachine
A (σ): { u,z,t*,w*
? [$(\gamma, y, \alpha) \rightarrow (\gamma, z, \alpha)$]
+ [$(\gamma, z, \alpha) \rightarrow (\gamma, t, \sigma\alpha)$]
– [$(\gamma, y, \alpha) \rightarrow (\gamma, z, \alpha)$]
+ [$(\gamma, y, \alpha) \rightarrow (\gamma, t, \alpha)$]
? [$(\gamma, x, \alpha) \rightarrow (\gamma, u, \alpha)$]
+ [$(\gamma, u, \sigma\alpha) \rightarrow (\gamma, w, \alpha)$]
– [$(\gamma, x, \alpha) \rightarrow (\gamma, u, \alpha)$]
+ [$(\gamma, x, \alpha) \rightarrow (\gamma, w, \alpha)$] }
B (σ): { v,r,s
A (σ)
? [$(\gamma, m, \alpha) \rightarrow (\gamma, s, \alpha)$]
– [$(\gamma, m, \alpha) \rightarrow (\gamma, s, \alpha)$]
? [$(\gamma, s, \sigma\alpha) \rightarrow (\gamma, r, \alpha)$]
+ [$(\gamma, m, \alpha) \rightarrow (\gamma, r, \alpha)$] }
C ( ): { n,t,p,v
– [$(\gamma, n, v\alpha) \rightarrow (\gamma, p, \alpha)$]
? [$(\gamma, m, \alpha) \rightarrow (\gamma, n, \alpha)$]
+ [$(\gamma, n, \alpha) \rightarrow (\gamma, ef1, \alpha)$]
? [$(\gamma, y, \alpha) \rightarrow (\gamma, t, \alpha)$]
+ [$(\gamma, t, \alpha) \rightarrow (\gamma, e3, \alpha)$] }

This example shows how adaptive automata may be employed to efficiently represent solutions for complex problems. The behavior of adaptive automata as piecewise finite-state- or structured pushdown-automata render them easy to understand and very adequate as implementation models. Many other classical subjects are as well effectively handled by adaptive automata whose behavior may often be far better, in both space and time aspects, than usual equivalents.

Such results encourage further efforts on the subject of this research. Parallel works are in progress at our institution exploring adaptive automata as programming paradigms, as computation models, as language implementation models, etc. In a near future we expect to have a full working high-level adaptive-paradigm language system based exclusively on adaptive automata, starting from its grammatical conception, including context-dependent aspects, run-time environment and the semantics of its dynamic behavior.

Acknowledgement. We would like to acknowledge the referees for their valuable suggestions, some of which have been included in this text.

References

[1] Shutt, J.N.- Self-modifying finite automata - Power and limitations. Technical Report WPI-CS-TR-95-4. Worcester Polytechnic Institute, Worcester, Massachusetts, Dec. 1995

[2] José Neto, J. - Adaptive Automata for context-sensitive languages – ACM SIGPLAN Notices, Nov. 1994

[3] José Neto, J. - Contribuições para a metodologia de construção de compiladores - Thesis - Escola Politécnica da Univ. de S. Paulo - 1990 (in Portuguese)

Author Index