Lecture Notes in Computer Science 2494

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

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

Bruce W. Watson Derick Wood (Eds.)

Implementation and Application of Automata

6th International Conference, CIAA 2001
Pretoria, South Africa, July 23-25, 2001
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

Bruce W. Watson
University of Pretoria, Department of Computer Science
Lynwood Road, Pretoria 0002, South Africa
E-mail: bwatson@cs.up.ac.za

Derick Wood
Hong Kong University of Science and Technology
Department of Computer Science
Clearwater Bay, Kowloon, Hong Kong
E-mail: dwood@cs.ust.hk

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743
ISBN 3-540-00400-9 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

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

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN: 10870693 06/3142 5 4 3 2 1 0

Foreword

The Sixth International Conference on Implementation and Application of Automata (CIAA 2001) — the first one held in the southern hemisphere — was held at the University of Pretoria in Pretoria, South Africa, on 23–25 July 2001.

This volume of Springer's Lecture Notes in Computer Science contains all the papers (including the invited talk by Gregor v. Bochmann) that were presented at CIAA 2001, as well as an expanded version of one of the poster papers displayed during the conference.

The conference addressed the issues in automata application and implementation. The topics of the papers presented in 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 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 of automata 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 cannot simply utilize the existing models and algorithms in automata theory to solve their problems. New models, or modifications of the existing models, are needed to satisfy their requirements. Also, the sizes of the typical problems in many of the new applications are astronomically larger than those used in the traditional applications. New algorithms and new representations of automata are required to reduce the time and space requirements of the computation.

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

We thank all of the program committee members and referees for their efforts in refereeing and selecting papers. This volume was edited with much help from Nanette Saes and Hanneke Driever, while the conference itself was run smoothly with the help of Elmarie Willemse, Nanette Saes, and Theo Koopman.

We also wish to thank the South African NRF (for funding airfares) and the Department of Computer Science, University of Pretoria, for their financial and logistic support of the conference. We also thank the editors of the Lecture Notes in Computer Science series and Springer-Verlag, in particular Anna Kramer, for their help in publishing this volume.

October 2002

Bruce W. Watson
Derick Wood

CIAA 2001 Program Committee

Bernard Boigelot	Université de Liege, Belgium
Jean-Marc Champarnaud	Université de Rouen, France
Maxime Crochemore	University of Marne-la-Vallée, France
Oscar Ibarra	University of California at Santa Barbara, USA
Lauri Karttunen	Xerox Palo Alto Research Center, USA
Nils Klarlund	AT&T Laboratories, USA
Denis Maurel	Université de Tours, France
Mehryar Mohri	AT&T Laboratories, USA
Jean-Eric Pin	Université Paris 7, France
Kai Salomaa	Queen's University, Canada
Helmut Seidl	Trier University, Germany
Bruce Watson (Chair)	University of Pretoria, South Africa Eindhoven University, The Netherlands
Derick Wood (Co-chair)	Hong Kong University of Science and Technology, China
Sheng Yu	University of Western Ontario, Canada

Table of Contents

Using Finite State Technology in Natural Language Processing of Basque ... 1
Iñaki Alegria, Maxux Aranzabe, Nerea Ezeiza, Aitzol Ezeiza, and Ruben Urizar

Cascade Decompositions are Bit-Vector Algorithms ... 13
Anne Bergeron and Sylvie Hamel

Submodule Construction and Supervisory Control: A Generalization ... 27
Gregor v. Bochmann

Counting the Solutions of Presburger Equations without Enumerating Them ... 40
Bernard Boigelot and Louis Latour

Brzozowski's Derivatives Extended to Multiplicities ... 52
Jean-Marc Champarnaud and Gérard Duchamp

Finite Automata for Compact Representation of Language Models in NLP ... 65
Jan Daciuk and Gertjan van Noord

Past Pushdown Timed Automata ... 74
Zhe Dang, Tevfik Bultan, Oscar H. Ibarra, and Richard A. Kemmerer

Scheduling Hard Sporadic Tasks by Means of Finite Automata and Generating Functions ... 87
Jean-Philippe Dubernard and Dominique Geniet

Bounded-Graph Construction for Noncanonical Discriminating-Reverse Parsers ... 101
Jacques Farré and José Fortes Gálvez

Finite-State Transducer Cascade to Extract Proper Names in Texts ... 115
Nathalie Friburger and Denis Maurel

Is this Finite-State Transducer Sequentiable? ... 125
Tamás Gaál

Compilation Methods of Minimal Acyclic Finite-State Automata for Large Dictionaries ... 135
Jorge Graña, Fco. Mario Barcala, and Miguel A. Alonso

Bit Parallelism – NFA Simulation ... 149
Jan Holub

Improving Raster Image Run-Length Encoding Using Data Order ... 161
Markus Holzer and Martin Kutrib

Enhancements of Partitioning Techniques
for Image Compression Using Weighted Finite Automata 177
Frank Katritzke, Wolfgang Merzenich, and Michael Thomas

Extraction of ϵ-Cycles from Finite-State Transducers 190
André Kempe

On the Size of Deterministic Finite Automata 202
Bořivoj Melichar and Jan Skryja

Crystal Lattice Automata 214
Jim Morey, Kamran Sedig, Robert E. Mercer, and Wayne Wilson

Minimal Adaptive Pattern-Matching Automata
for Efficient Term Rewriting 221
Nadia Nedjah and Luiza de Macedo Mourelle

Adaptive Rule-Driven Devices - General Formulation and Case Study 234
João José Neto

Typographical Nearest-Neighbor Search in a Finite-State Lexicon
and Its Application to Spelling Correction 251
Agata Savary

On the Software Design of Cellular Automata Simulators
for Ecological Modeling 261
Yuri Velinov

Random Number Generation with $\oplus$-NFAs 263
Lynette van Zijl

Supernondeterministic Finite Automata 274
Lynette van Zijl

Author Index 289

Using Finite State Technology in Natural Language Processing of Basque

Iñaki Alegria, Maxux Aranzabe, Nerea Ezeiza, Aitzol Ezeiza, and Ruben Urizar

Ixa taldea, University of the Basque Country, Spain
i.alegria@si.ehu.es

Abstract. This paper describes the components used in the design and implementation of NLP tools for Basque. These components are based on finite state technology and are devoted to the morphological analysis of Basque, an agglutinative pre-Indo-European language. We think that our design can be interesting for the treatment of other languages. The main components developed are a general and robust morphological analyser/generator and a spelling checker/corrector for Basque named Xuxen. The analyser is a basic tool for current and future work on NLP of Basque, such as the lemmatiser/tagger Euslem, an Intranet search engine or an assistant for verse-making.

1 Introduction

This paper describes the components used in the design and implementation of NLP tools for Basque. These components are based on finite state technology and are devoted to the morphological analysis of Basque, an agglutinative pre-Indo-European language. We think that our design can be interesting for the treatment of other languages.

The main components developed are a general and robust morphological analyser/generator (Alegria et al. 1996) and a spelling checker/corrector for Basque named Xuxen (Aldezabal et al. 1999). The analyser is a basic tool for current and future work on NLP of Basque, for example the lemmatiser/tagger *Euslem* (Ezeiza et al. 1998), an Intranet search engine (Aizpurua et al. 2000) or an assistant for verse-making (Arrieta et al. 2000)

These tools are implemented using lexical transducers. A lexical transducer (Karttunen 1994) is a finite-state automaton that maps inflected surface forms to lexical forms, and can be seen as an evolution of two-level morphology (Koskenniemi 1983; Sproat 1992) where the use of diacritics and homographs can be avoided and the intersection and composition of transducers is possible. In addition, the process is very fast and the transducer for the whole morphological description can be compacted in less than one Mbyte. The tool used for the implementation is the *fst* library of *Inxight*[1] (Karttunen and Bessley 1992; Karttunen 1993; Karttunen et al. 1996). Similar compilers have been developed by other groups (Mohri 1997; Daciuk et al. 1998).

[1] Inxight Software, Inc., a Xerox Enterprise Company (www.inxight.com)

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 1-12, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 The Design of the Morphological Analyser

The design that we propose was carried out because after testing different corpora of Basque the coverage was just about 95%. This poor result was due (al least partially) to the recent standardisation and the widespread dialectal use of Basque. In order to improve the coverage, we decided that it was necessary to manage non-standard uses and forms whose lemmas were not in the lexicon[2], if we wanted to develop a comprehensive analyser. So three different ways were proposed: management of user's lexicon, analysis of linguistic variants and analysis without lexicon.

We propose a multilevel method, which combines robustness and avoiding of overgeneration in order to build a general-purpose morphological analyser/generator. Robustness is basic in corpus-analysis but sometimes to obtain it overgeneration is produced. Overgeneration increases ambiguity and many times this ambiguity is not real and causes poor results (low precision) in applications based on morphology such as spelling correction, morphological generation and tagging.

The design we propose for robustness without overgeneration consists of three main modules (Fig. 1):

1. The standard analyser using general and user's lexicons. This module is able to analyse/generate standard language word-forms. In our applications for Basque we defined —using a database— about 70,000 entries in the general lexicon, more than 130 patterns of morphotactics and two rule systems in cascade, the first one for long-distance dependencies among morphemes and the second for morphophonological changes. The three elements are compiled together in the *standard transducer*. To deal with the user's lexicon the general transducer described below is used.
2. The analysis and normalization of linguistic variants (dialectal uses and competence errors). Because of non-standard or dialectal uses of the language and competence errors, the standard morphology is not enough to offer good results when analysing real text corpora. This problem becomes critical in languages like Basque in which standardisation is in process and dialectal forms are still of widespread use. For this process the standard transducer is extended producing the *enhanced transducer*.
3. The guesser or analyser of words without lemmas in the lexicons. In this case the standard transducer is simplified removing the lexical entries in open categories (names, adjectives, verbs, …), which constitute the vast majority of the entries, and is substituted by a general automata to describe any combination of characters. So, the *general transducer* is produced combining this general lemma-set with affixes related to open categories and general rules.

Important features of this design are homogeneity, modularity and reusability because the different steps are based on lexical transducers, far from ad-hoc solutions, and these elements can be used in different tools.

---

[2] In some systems lemmas corresponding to unknown words are added to the lexicon in a previous step, but if we want to built a robust system this is not be acceptable.

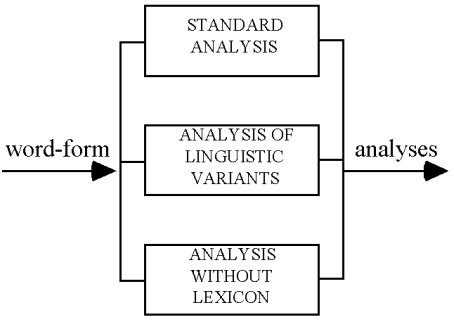


Fig. 1. Design of the analyser

This can be seen as a variant of constraint relaxation techniques used in syntax (Stede 1992), where the first constraint demands standard language, the second one combines standard and linguistic variants, and the third step allows free lemmas in open categories. Only if the previous steps fail the results of the next step are included in the output. Relaxation techniques are used in morphology also by Oflazer (Oflazer 1996) but in a different way[3]. With this design the obtained coverage is 100% and precision up to 99%.

The combination of three different levels of analysis and the design of the second and third levels are original as far as we know.

3 The Transducers

A lexical transducer (Karttunen 1994) is a finite-state automaton that maps inflected surface forms to lexical forms, and can be seen as an evolution of the two-level morphology where:

- Morphological categories are represented as part of the lexical form. Thus, diacritics may be avoided.
- Inflected forms of the same word are mapped to the same canonical dictionary form. This increases the distance between the lexical and surface forms. For instance *better* is expressed through its canonical form *good* (*good+COMP:better*).
- Intersection and composition of transducers is possible (Kaplan and Kay 1994). In this way the integration of the lexicon, which will be another transducer, can be solved in the automaton and the changes between lexical and surface level can be expressed as a cascade of two-level rule systems where, after the intersection of the rules, the composition of the different levels is carried out (Fig. 2).

[3] He uses the term *Error-tolerant morphological analysis* and says: "*The analyzer first attempts to parse the input with t=0, and if it fails, relaxes t ...*"

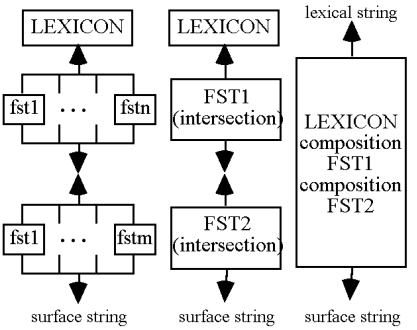


Fig. 2. Intersection and composition of transducers (from Karttunen et al. 1992)

3.1 The Standard Transducer

Basque is an agglutinative language, that is, for the formation of words the dictionary entry independently takes each of the elements necessary for the different functions (syntactic case included). More specifically, the affixes corresponding to the determinant, number and declension case are taken in this order and independently of each other (deep morphological structure). One of the main characteristics of Basque is its declension system with numerous cases, which differentiates it from the languages spoken in the surrounding countries.

We have applied the two-level model but combining the following transducers:

1. *FST1* or Lexicon. Over 70,000 entries have been defined corresponding to lemmas and affixes, grouped into 170 sublexicons. Each entry of the lexicon has, in addition to the morphological information, its continuation class, which is made up of a group of sublexicons. Lexical entries, sublexicons and continuation classes all together define the morphotactics graph, i.e. the automaton that describes the lexical level. The lexical level will be the result of the analysis and the source for the generation. This description is compiled and minimized in a transducer with 1.5 million states and 1.6 million arcs. The upper side of the transducer is the whole morphological information, and the lower side is composed of the morphemes and the minimal morphological information to control the application of the other transducers in cascade (FST2 and FST3).
2. *FST2*: Constraint of long-distance dependencies. Some dependencies among morphemes can be expressed with continuation classes because co-occurrence restrictions exist between morphemes that are physically separated in a word (Bessley 1998). For instance, in English, *en-*, *joy* and *-able* can be linked together (*enjoyable*), but it is not possible to link only *joy* and *–able (joyable*)*. Using morphophonological rules is a simple way to solve them when, as in our system, it is only necessary to ban some combinations.
   Three rules have been written to solve long-distance dependencies of morphemes: one in order to control hyphened compounds, and two so as so avoid both prefixed and suffixed causal conjunctions (*bait-* and *–lako*) occurring together

(*baitielako**). These rules have been put in a different rule system closer to the lexical level, without mixing morphotactics and morphophonology. The transducer is very small: 26 states and 161 arcs.

3. *FST3*: set of morphophonological rules. 24 two-level rules have been defined to express the morphological, phonological and orthographic changes between the lexical and the surface levels that happen when the morphemes are combined. Details about these rules can be consulted in (Alegria et al. 1996) The transducer is not very big but it is quite complex. It is composed of 1,300 states and 19,000 arcs.

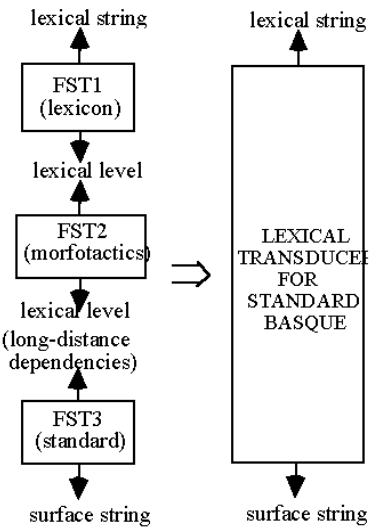


Fig. 3. Cascade of three transducers for standard analysis

The three transducers are combined by composition to build the standard analyser, which attaches to each input word-form all possible interpretations and its associated information. The composed transducer has 3.6 millions states and 3.8 million arcs, but is minimized into 1.9 M-states and 2 M-arcs, which take 3.2 Megabytes in disk.

A simple example of the language involved in the transducer is given in Fig. 4

3.2 The Enhanced Transducer

A second morphological subsystem, which analyses, normalizes, and generates linguistic variants, is added in order to increase the robustness of the morphological processor. This subsystem has three main components:

1. *FST1**: New morphemes linked to their corresponding standard ones in order to normalize or correct the non-standard morphemes are added to the standard lexicon. Thus, using the new entry *tikan*, dialectal form of the ablative singular morpheme, linked to its corresponding right entry *tik* will be able to analyse and correct word-forms such *etxetikan*, *kaletikan*,... (variants of *etxetik* 'from the

house', *kaletik* 'from the street', ...). More than 1500 additional morphemes have been included. Changes in the morphotactical information —continuation class— corresponding to some morphemes of the lexicon have been added too. In addition to this, the constraint of long-distance dependencies have been eliminated because sometimes these constraints are not followed, so FST2 is not applied. The compiled transducer for the enhanced lexicon increases the states from 1.5 to 1.6 millions and the arcs from 1.6 millions to 1.7..

2. *FST3**: The standard morphophonological rule-system with a small change: the morpheme boundary (+ character) is not eliminated in the lower level in order to use it to control changes in FST4. So, the language at this level correspond to the surface level enriched with the + character.
3. *FST4*: New rules describing the most likely regular changes that are produced in the linguistic variants. These rules have the same structure and management as the standard ones but all of them are optional. For instance, the rule `h:0 => V:V_V:V` describes that between vowels the `h` of the lexical level may disappear in the surface level. In this way the word-form *bear*, misspelling of *behar* (to need), can be analysed. As Fig. 5 shows, it is possible and clearer to put these non-standard rules in another level close to the surface, because most of the additional rules are due to phonetic changes and do not require morphological information.

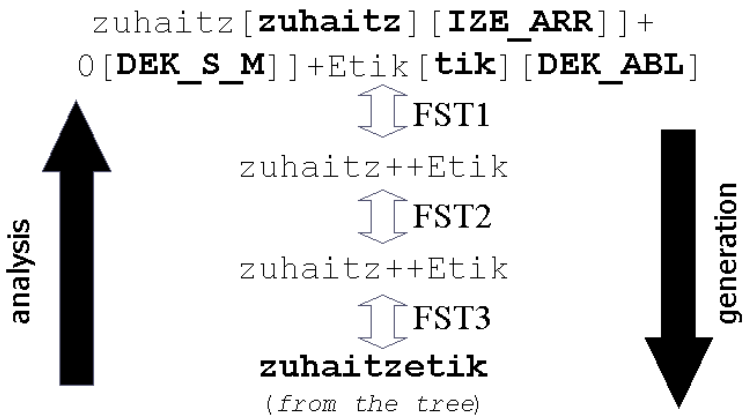


Fig. 4. Example of cascade of transducer for standard analysis[4]

The composition of the FST1* and FST3* is similar in the number of states and arcs to the standard transducer, but when FST4 is added the number of states increases

[4] *IZE_ARR*: common noun, *DEK_S_M* singular number, *Etik tik* suffix with epenthetical e, *DEK_ABL*: ablative declension case.
A rule in FST3 controls the realization of the epenthetical e (the next rule is a simplification): `E:e <=> Cons +: +: _` It can be read as "the epenthetical e is realized as e after a consonant in the previous morpheme".
zuhaitzetik: from the tree

from 3.7 million states to 12 millions and the number of arcs from 3.9 millions to 13.1 millions. Nevertheless, it is minimized into 3.2 M-states and 3.7 M-arcs, which takes 5.9 Megabytes in disk.

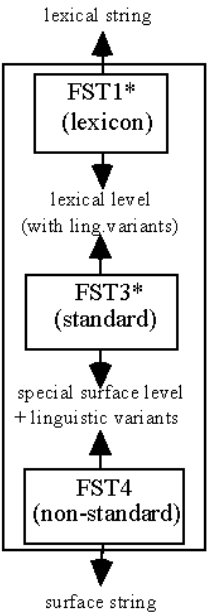


Fig. 5. Cascade of three transducers in the enhanced subsystem

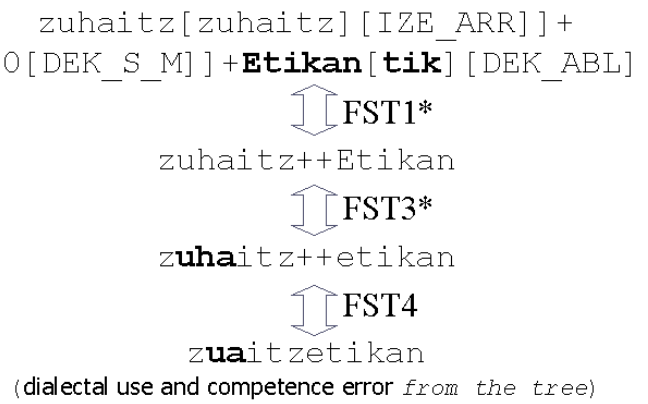


Fig. 6. Example of cascade of transducer for non-standard analysis[5]

[5] *Zuaitzetikan*: variation of *zuhaitzetik* (from the tree) with two changes: dropped h and dialectal use of *tikan*.

3.3 The General Transducer

The problem of unknown words does not disappear with the previous transducer. In order to deal with it, a general transducer has been designed to relax the need of lemmas in the lexicon. This transducer was initially (Alegria et al. 1997) based on the idea used in speech synthesis (Black et al. 1991) but now it has been simplified. Daciuk (Daciuk 2000) proposes a similar way when he describes the *guessing automaton*, but the construction of the automaton is more complex.

The new transducer is the standard one modified in this way: the lexicon is reduced to affixes corresponding to open categories[6] and generic lemmas for each open category, while standard rules remain. So, the standard rule-system (FST3) is composed of a mini-lexicon (FST0) where the generic lemmas are obtained as a result of combining alphabetical characters and can be expressed in the lexicon as a cyclic sublexicon with the set of letters (some constraints are used with capital/non-capital letters according to the part of speech). In fig. 7 the graph corresponding to the mini-lexicon (FST0) is shown.

The composed transducer is tiny, it is into 8,5 thousand states and 15 thousand arcs. Each analysis in the result is a possible lemma with the whole morphological information corresponding to the lemma and the affixes.

This transducer is used in two steps of the analysis: in the standard analysis and in the analysis without lexicon (named guessing in taggers).

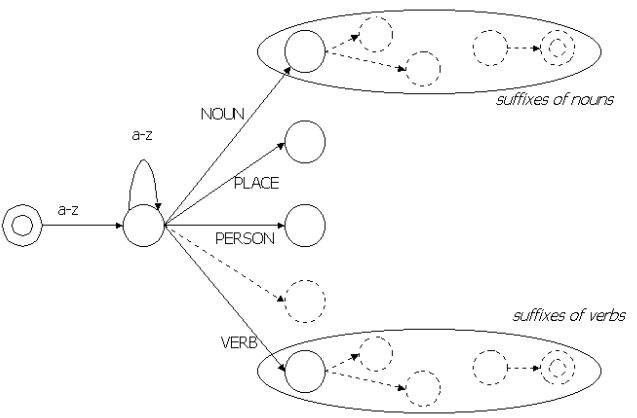


Fig. 7. Simplified graph of the mini-lexicon

In order to avoid the need of compiling the user's lexicon with the standard description, the general transducer is used in the standard analysis, and if the hypothetical lemma is found in the user's lexicon the analysis is added to the results obtained in the standard transducer.

If no results are obtained in the standard and enhanced steps the results of the general transducer will be the output of the general analyser.

---

[6] There are seven open categories and the most important ones are: common nouns, personal names, place nouns, adjectives and verbs.

3.4 Local Disambiguation and Ongoing Work

Although one of the targets in the designed system is to avoid overgeneration, in the enhanced and general transducers overgeneration can still be too high for some applications.

Sometimes, the enhanced transducer returns analyses for words the lemmas of which are not included in the lexicon. That is to say, words that are not variants are analysed as such. Bearing in mind that the transducer is the result of the intersection of several rules each one corresponding to an optional change, the resulting transducer permits all the changes to be done in the same word. However, some combinations of changes seldom occur, so it is the general transducer that must accomplish the analysis.

Besides, sometimes there is more than one analysis as variant and it is necessary to choose among them. For example, analysing the word-form *kaletikan* (dialectal form) two possible analysis are obtained: *kale+tik* (from the street) and *kala+tik* (from the cove), but the first analysis is more probable because only one change has been done.

The solution could be to use a probabilistic transducer (Mohri 1997), or to improve the tool in order to obtain not only the lexical level but also the applied rules (this is not doable with the tools we have).

Currently, we use a local disambiguator that calculates the edit distance between the analysed word and each possible normalized word (generated using standard generation), choosing the most standard one(s) i.e. those with the lowest edit distance. Above a threshold, the results of this transducer are discarded. In the example above, *kaletikan* is compared to *kaletik* and *kalatik* (surface level of *kale+tik* and *kala+tik*). *kaletik* is chosen because its distance from *kaletikan* is shorter (2) than that of *kalatik*.

The general transducer presents two main problems:

- too many different tags can be produced. However, this problem is solved by a context based disambiguator (Ezeiza et al. 1998)
- multiple lemmas for the same or similar morphological analysis. This is a problem when we want to built a lemmatizer. For example if *bitaminiko* (vitaminic) is not in the lexicon the results analysing *bitaminikoaren* (from the vitaminic) as adjective can be multiple: *bitamini+ko+aren*, *bitaminiko+aren* and *bitaminikoaren*, but the only right analysis is the second.

In the first case information about capital letters and periods is used to accept/discard some tags, but the second case is the main problem for us. A probabilistic transducer for the sublexicon with the set of letter-combinations would be a solution. However, for the time being, heuristics using statistics about final trigrams (of characters) in each category, cases, and lenght of lemmas are used to disambiguate the second case.

4 The Spelling Checker/Corrector

The three transducers are also used in the spelling checker/corrector but, in order to reduce the use of memory, most of the morphological information is eliminated.

The spelling checker accepts as correct any word that allows a correct standard morphological analysis. So, if the standard transducer returns any analysis (the word is standard) or one of the possible lemmas returned by the general transducer is in the user's lexicon, the word is accepted.

Otherwise, a misspelling is assumed and the user gets a warning message and is given different options. One of most interesting option given is to include the lemma of the world in the user's lexicon. From then on, any inflected and derived form of this lemma will be accepted without recompiling the transducer.

For this purpose the system has an interface, in which the part of speech must be specified along with the lemma when adding a new entry to the user lexicon.

The proposals given for a misspelled word are divided in two groups: competence errors and typographical errors. Although there is wide bibliography about the correction problem (Kukich 1992), most of the authors do not mention the relation between them and morphology. They assume that there is a whole dictionary of words or that the system works without lexical information. Oflazer and Guzey (1994) faced the problem of correcting words in agglutinative languages. Bowden and Kiraz (Bowden and Kiraz 1995) applied morphological rules in order to correct errors in nonconcatenative phenomena.

The need of managing competence errors —also named orthographic errors— has been mentioned and reasoned by different authors (van Berkel and de Smedt 1988) because this kind of errors are said to be more persistent and make a worse impression. When dealing with the correction of misspelled words the main problem faced was that, due to the recent standardisation and the widespread dialectal use of Basque, competence errors or linguistic variants were more likely and therefore their treatment became critical.

When a word-form is not accepted it is checked against the enhanced transducer. If the incorrect form is now recognised—i.e. it contains a competence error— the correct lexical level form is directly obtained and, as the transducers are bi-directional, the corrected surface form will be generated from the lexical form using the standard transducer.

For instance, in the example above, the word-form `beartzetikan` (misspelling of `behartzetik` "from the need") can be corrected although the edit distance is three. The complete process of correction would be the following:

- Decomposition into three morphemes: *behar* (using a rule to guess the h), *tze* and *tikan*.
- *tikan* is a non-standard use of *tik* and as, they are linked in the lexicon, this is the chosen option.
- The standard generation of *behar+tze+tik* obtains the correct word *behartzetik*.

The treatment of typographical errors is quite conventional and only uses the standard transducer to test hypothetical proposals. It performs the following steps:

- Generating hypothetical proposals to typographical errors using Damerau's classification.
- Spelling checking of proposals.

The results are very good in the case of competence errors —they could be even better if the non-standard lexicon was improved — and not so good for typographical

errors. In the last case, only errors with an edit distance of one have been planned. It would be possible to generate and test all the possible words with a higher edit distance, but the number of proposals would be very big. We are planning to use the Oflazer and Guzey's proposal, which is based on flexible morphological decomposition.

5 Conclusions

In this paper we have presented an original methodology that allows combining different transducer to increase the coverage and precision of basic tools for NLP of Basque. The design of the enhanced and general transducers that we propose is new as far as we know. We think that our design could be interesting for the robust treatment of other languages

Acknowledgements

This work has had partial support from the Education Department of the Government of the Basque Country **(**reference UE1999-2**)**. We would like to thank Xerox for allowing us to use their tools, and also Lauri Karttunen for his help.
Thanks to anonymous referees for helping us improving the paper.

References

[1] Aizpurua I, Alegria I, Ezeiza N (2000) GaIn: un buscador Internet/Intranet avanzado para textos en euskera. Actas del XVI Congreso de la SEPLN Universidad de Vigo.

[2] Aldezabal I, Alegria I, Ansa O, Arriola JM, Ezeiza N (1999) Designing spelling correctors for inflected languages using lexical transducers. Proceedings of EACL'99, 265-266. Bergen, Norway.

[3] Alegria I, Artola X, Sarasola K, Urkia M (1996) Automatic morphological analysis of Basque. Literary and Linguistic Computing vol. 11, No. 4, 193-203. Oxford University Press. Oxford.

[4] Alegria I, Artola X, Ezeiza N, Gojenola K, Sarasola K (1996) A trade-off between robustness and overgeneration in morphology. Natural Language Processing and Industrial Applications. vol I pp 6-10. Moncton, Canada.

[5] Alegria I, Artola X, Sarasola K (1997) Improving a Robust Morphological Analyser using Lexical Transducers. Recent Advances in Natural Language Processing. *Current Issues in Linguistic Theory (CILT) series*. John Benjamins publisher company. vol. 136. pp 97-110.

[6] Arrieta B, Arregi X, Alegria I (2000) An Assistant Tool For Verse-Making In Basque Based On Two-Level Morphology. Proceedings of ALLC/ACH 2000 . Glasgow, UK.

[7] Bessley K (1998) Constraining Separated Morphotactic Dependencies in Finite State Grammars. Proc. of the International Workshop on Finite State Methods in NLP. Ankara.

[8] Black A, van de Plassche J, Williams B (1991) Analysis of Unkown Words through Morphological Descomposition. Proc. of 5th Conference of the EACL, vol. 1, pp 101-106.
[9] Bowden T, Kiraz G (1995) A morphographemic model for error correction in non-concatenative strings. Proc. of the 33rd Conference of the ACL, pp 24-30.
[10] Daciuk J, Watson B, Watson R (1998) Incremental Construction of Minimal Acyclic Finite State Automata and Transducers. Proc. of the International Workshop on Finite State Methods in NLP. Ankara.
[11] Daciuk J (2000) Finite State Tools for Natural Language Processing. Proceedings of the COLING 2000 workshop Using Toolsets and Architectures to Build NLP Systems, Luxembourg.
[12] Ezeiza N, Aduriz I, Alegria I, Arriola JM, Urizar R (1998) Combining Stochastic and Rule-Based Methods for Disambiguation in Agglutinative Languages. COLING-ACL'98, Montreal (Canada).
[13] Kaplan RM and Kay M (1994) Regular models of phonological rule systems. Computational Linguistics, vol.20(3): 331-380. 1994.
[14] Karttunen L (1993) Finite-State Lexicon Compiler. Xerox ISTL-NLTT-1993-04-02.
[15] Karttunen L (1994) Constructing Lexical Transducers, Proc. of COLING´94, 406-411.
[16] Karttunen L (2000) Applications of Finite-State Transducers in Natural Language Processing. Proceedings of CIAA-2000. Lecture Notes in Computer Science. Springer Verlag.
[17] Karttunen L and Beesley KR (1992) Two-Level Rule Compiler. Xerox ISTL-NLTT-1992-2.
[18] Karttunen L, Kaplan RM, Zaenen (1992) A Two-level morphology with composition. Proc. of COLING´92. 1992.
[19] Karttunen L, Chanod JP, Grenfenstette G, Schiller A (1996) Regular Expressions for Language Engineering. Natural Language Engineering, 2(4): 305:328.
[20] Koskenniemi, K (1983) Two-level Morphology: A general Computational Model for Word-Form Recognition and Production, University of Helsinki, Department of General Linguistics, Publications, 11.
[21] Kukich K (1992) Techniques for automatically correcting word in text. ACM Computing Surveys, vol.24, No. 4, 377-439.
[22] Mohri, M (1997) Finite-state transducers in language and speech processing. Computational Linguistics 23(2):269-322.
[23] Oflazer K (1996) Error-tolerant Finite State Recognition with Applications to Morphological Analysis and Spelling Correction. Computational Linguistics 22(1):73-89.
[24] Oflazer K, Guzey C. (1994). Spelling Correction in Aglutinative Languages, Proc. of the ANLP-94, Sttutgart.
[25] Sproat R (1992) Morphology and Computation. The MIT Press.
[26] Stede M (1992) The Search of Robustness in Natural Language Understanding. Artificial Intelligence Review 6, 383-414. 1992.
[27] Van Barkel B, De Smedt K (1988) Triphone analysis: a combined method for the correction of orthographic and typographical errors. Proceedings of the Second Conference ANLP (ACL), pp 77-83.

Cascade Decompositions are Bit-Vector Algorithms

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. A vector algorithm is an algorithm that applies a bounded number of vector operations to an input vector, regardless of the length of the input. In this paper, we describe the links between the existence of vector algorithms and the cascade decompositions of counter-free automata.

We show that any computation that can be carried out with a counter-free automaton can be recast as a vector algorithm. Moreover, we show that for a class of automata that is closely related to algorithms in bio-computing, the complexity of the resulting algorithms is linear in the number of transitions of the original automaton.

1 Introduction

The goal of this paper is to investigate the links between the *Krohn-Rhodes Theorem* [3], and the so-called *bit-vector algorithms* that popped up recently in the field of bio-computing to accelerate the detection of similarities between genetic sequences [6].

A vector algorithm is an algorithm that applies a bounded number of vector operations to an input, regardless of the length of the input. These algorithms can thus be implemented in parallel, and/or with bit-wise operations available in processors, leading to highly efficient computations.

These algorithms are usually derived from an input-output automaton that models a computation, but they often use specific properties of its transition table in order to produce an efficient algorithm. It is thus natural to ask whether there is a general way to construct them. In [1], we identified a class of automata, the *solvable* automata, for which we could prove the existence of bit-vector algorithms.

This paper extends our previous work in two directions. We first extend the construction of bit-vector algorithms to the class of *counter-free* automata. Drawbacks of this construction, which relies on the cascade decomposition of the automata, are that there is no easy way to obtain it, and that the complexity of the resulting algorithms can be exponential in the number of transitions [5]. Still, the second, and surprising result, is that any solvable automaton admits a bit-vector algorithm whose complexity is *linear* in the number of transitions.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 13–26, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 What is a (Bit) Vector Algorithm?

A *vector* algorithm is an algorithm which, on input vector $\boldsymbol{e} = (e_1 e_2 \dots e_m)$, computes an output vector $\boldsymbol{r} = (r_1 r_2 \dots r_m)$ in a bounded number of steps, *independent* of m. Each step of the computation consists on applying, componentwise, a single operation on the input vector. We talk of *bit-vector* algorithms when the operations are restricted to bit-wise operations such as logical operators, denoted by the usual symbols $\neg$, $\vee$, and $\wedge$; binary addition; and *shifts* that are defined, for $\boldsymbol{e} = (e_1 \dots e_m)$ as

$$\uparrow_v \boldsymbol{e} = (v e_1 \dots e_{m-1}).$$

Here the values of $\boldsymbol{e}$ have been shifted to the right, and the first component is set to v. As a running example, consider the following automaton.

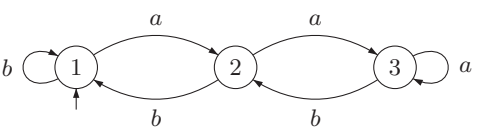


Fig. 1. A bounded counter

Given an input word $\boldsymbol{e} = (e_1 e_2 \dots e_m)$, we are interested in the sequence of output states. The standard way of carrying this computation is to visit the states of the automaton using the input word, which is a procedure whose complexity is proportional to the length of $\boldsymbol{e}$. On the other hand, the following surprising formula decides whether the output state is 1, in 8 operations.

$$\boldsymbol{b} \wedge (\uparrow_1 \boldsymbol{b} \vee ((\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) + \neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b})) \tag{1}$$

Where $\boldsymbol{a}$ and $\boldsymbol{b}$ stand respectively for the characteristic bit-vector of the letters a and b in $\boldsymbol{e}$, that is:

$$\begin{aligned} \boldsymbol{a}_i &= 1 \ \text{ iff } \ e_i = a \\ \boldsymbol{b}_i &= 1 \ \text{ iff } \ e_i = b. \end{aligned}$$

For example, if $\boldsymbol{e} = (baababbb)$ then

$$\begin{aligned} \boldsymbol{a} &= (01101000) \\ \boldsymbol{b} &= (10010111). \end{aligned}$$

Computing Formula (1) with these values yield:

$$\neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b}) = \neg((11001011) \wedge (10010111)) = (01111100),$$

and

$$\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a} = (00110100) \wedge (01101000) = (00100000),$$

thus,

$$((\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) + \neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b})) = (00100000) + (01111100) = (01000010),$$

with the binary addition carried from left to right, and Formula (1) is:

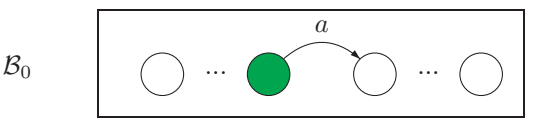
$$\begin{aligned}(b) \wedge (\uparrow_1 \boldsymbol{b} \vee ((\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) + \neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b})) &= (10010111) \wedge ((11001011) \vee \\ &\quad (01000010)) \\ &= (10000011).\end{aligned}$$

Formula (1) requires 5 logical operations, 2 shifts, and 1 binary addition with carry. This formula can thus be computed very efficiently, and the number of steps in the computation is independent of the length of the input word $\boldsymbol{e}$. The true agenda of this paper is to fully understand the correspondence between the bounded counter automaton of Fig.1 and Formula (1).

3 Cascades Made Simple

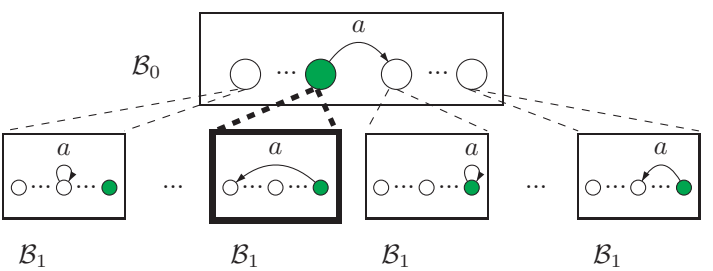
The *cascade* product [8], and its more algebraic counterpart, the *wreath* product, have awkward definitions that contributed greatly to their almost total neglect from the computer science community. However, in the next pages, we will try to give the essential flavor of the construction in simple terms, while – let's hope – giving enough details to document the links with bit-vector algorithms.

Consider an automaton $\mathcal{B}_0$, with n states, which we will represent as the following generic box,

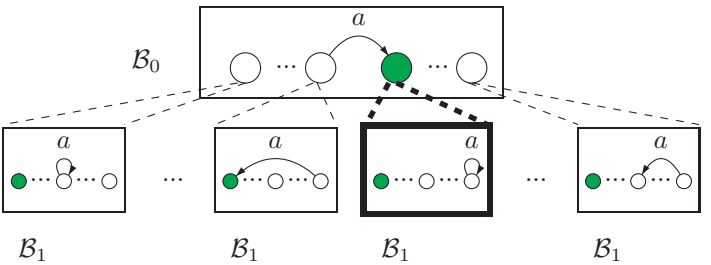


where a stands for an arbitrary transition. In order to define the cascade product $\mathcal{B}_0 \circ \mathcal{B}_1$, we attach to each state of $\mathcal{B}_0$ a clone of an automaton $\mathcal{B}_1$, with m states, and whose transition function may *vary* among different copies. That is, automaton $\mathcal{B}_1$ has possibly n different transition functions.

The whole device operates with the following protocol. Automaton $\mathcal{B}_0$ has a current state – in gray –, and each version of automaton $\mathcal{B}_1$ has the same current state – also in gray. This pair of states is the *global state* of the device.



On input a, the behavior of $\mathcal{B}_0$ is the normal automaton behavior, and the behavior of $\mathcal{B}_1$ is given by the clone which is attached to the current state of $\mathcal{B}_0$. Assuming the above global state, on input a, the next state of the product would be the following.



Clearly, the global behavior can be described by an automaton with $n \times m$ states, since for each global state (q_1, q_2), and each input letter, there is a unique corresponding global state. This construction can also be iterated.

Formally, we have the following definition of the cascade product.

Definition 1. *A* cascade product $\mathcal{C} = (\Sigma, Q, \delta') = \mathcal{B}_0 \circ \mathcal{B}_2 \circ \ldots \circ \mathcal{B}_{n-1}$ *is a possibly incomplete automaton such that:*

1. *For all* i, $0 \leq i \leq n-1$, $\mathcal{B}_i = (Q_0 \times \cdots \times Q_{i-1} \times \Sigma, Q_i, \delta_i)$, *where* δ_i *is a partial transition function.*
2. $Q = Q_0 \times \ldots \times Q_{n-1}$ *and the global transition function is evaluated coordinate-wise according to*

$$\delta'(q_0 \ldots q_{n-1}, \sigma) = (\delta_0(q_0, \sigma), \ldots, \delta_{n-1}(q_0 \ldots q_{n-2}, \sigma))$$

The cascade decomposition of an automaton $\mathcal{A}$ can then be define as follows.

Definition 2. *Let* $\mathcal{A}$ *be an automaton. A* cascade decomposition *of* $\mathcal{A}$ *is given by a cascade product* $\mathcal{C} = \mathcal{B}_0 \circ \mathcal{B}_2 \circ \ldots \circ \mathcal{B}_{n-1}$, *and a (partial) homomorphism* φ *from* $\mathcal{C}$ *to* $\mathcal{A}$.

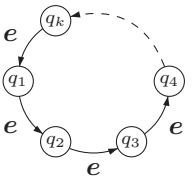
For example, the bounded counter of Fig. 1 admits the following decomposition, with the homomorphism:

$$\varphi(0,0) = 1 \quad \varphi(0,1) = \varphi(1,0) = 2 \quad \varphi(1,1) = 3.$$

In this example, the global behavior $\mathcal{C}$ of the cascade product, and the homomorphism φ, can be illustrated by the following graph:

The rewards of going trough such a definition is a theorem by Krohn and Rhodes (1965) that establishes that any automaton admits a cascade decomposition whose elements are very simple, and whose nature reflects deep algebraic properties of the language recognized by the automaton. Here we need a special case of this theorem which concerns *counter-free* automata.

In general, a word $\boldsymbol{e}$ induces a non-trivial permutation on the states of an automaton $\mathcal{A}$ if there is a sequence of $k > 1$ states in $\mathcal{A}$ that are mapped circularly by $\boldsymbol{e}$.



An automaton is *counter-free* if no word induces such a permutation. A transition a is a *reset* if the function induced on the states of $\mathcal{A}$ is constant. A *reset* automaton is an automaton in which all transitions are resets or identities.

Theorem 1. *(Krohn-Rhodes) Any counter-free automaton admits a cascade decomposition of binary reset automata.*

This theorem provides the link between counter-free automata and vector algorithms. Indeed, given a cascade decomposition $(\mathcal{C}, \varphi)$ of an automaton $\mathcal{A}$, it is easy to produce an elementary logical characterization of the output states of $\mathcal{A}$. For example, in the decomposition of Fig.3, we have the following:

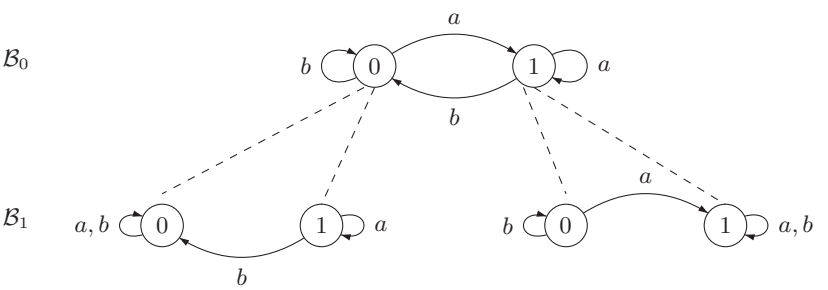


Fig. 2. A cascade decomposition

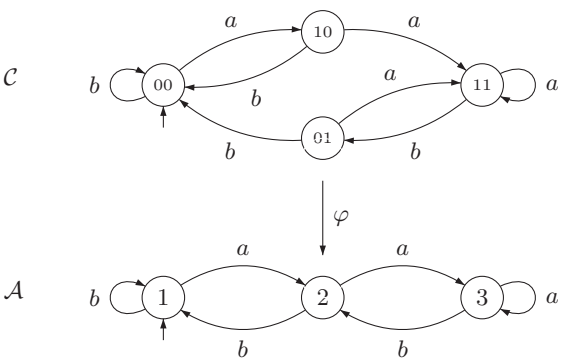


Fig. 3. The homomorphism from $\mathcal{C} = \mathcal{B}_0 \circ \mathcal{B}_1$ to $\mathcal{A}$

$\mathcal{A}$ is in state 1 iff
$\mathcal{C}$ is in state $(0, 0)$ iff
$\mathcal{B}_0$ is in state $0 \wedge \mathcal{B}_1$ is in state 0

In general, since the homomorphism φ is surjective, we have:

Corollary 1. *For any counter-free automata $\mathcal{A}$, the proposition "$\mathcal{A}$ is in state s" is equivalent to a disjunction of propositions of the form:*

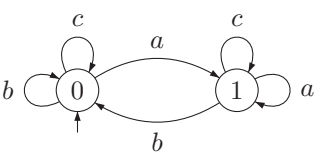
$$\mathcal{B}_0 \text{ is in state } s_0 \wedge ... \wedge \mathcal{B}_{n-1} \text{ is in state } s_{n-1}$$

where each $\mathcal{B}_i$ is a binary reset automaton.

Corollary 1 implies that the problem of translating a counter-free automata computation into bit-vector algorithms reduces to the problem of translating binary resets automata. This is the subject of the next section.

4 The Addition Lemma

A binary reset automaton has a very simple structure that is depicted by the following 'generic' binary reset automaton. Both letters a and b are resets to, respectively states 1 and 0, and the letter c is an identity.



Assume that state 0 is the initial state, and define L_q to be the set of non-empty words that *end* in state q. We have the following characterizations of L_0.

$$L_0 = \{\boldsymbol{e} |\ \boldsymbol{e} \text{ is } (c \dots c), \text{ or there is a } b \text{ in } \boldsymbol{e} \text{ and no } a \text{ since the last } b\}$$

Given a word $\boldsymbol{e}$, consider the characteristic bit-vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. We have the following lemma [1] which relates membership to L_0 to bit-vector operations, where the addition is the usual binary addition with carry propagation, performed from left to right. The proof is elementary, but it illustrates many of the techniques for manipulating bit vectors.

[1] Lemma 1 is strikingly similar to the past temporal logical formulas of [4]. Maler and Pnueli code the language L_q with the logical formula:

$$(\neg out_q)\mathcal{S}(in_q)$$

which can be loosely translated as "*there was no transition that went out of state q since the last reset transition to state q*". In the next section, we will use this similarity to discuss the complexity of vector algorithms.

Lemma 1. *(The Addition Lemma) The word* $\boldsymbol{e} \in L_0$ *if and only if the last bit of*

$$\boldsymbol{b} \vee (\neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b}))$$

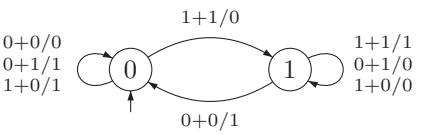
is set to 1.

Proof. Suppose that $\boldsymbol{e}$ is in L_0. If $\boldsymbol{e}$ is of the form $(c \dots c)$, then $\boldsymbol{a} = \boldsymbol{b} = (0 \dots 0)$, and we get easily that $\neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b}) = (1 \dots 1)$.

Now, suppose that there is a b in $\boldsymbol{e}$ and no occurrence of a since the last occurrence of b; suppose also that the last letter of $\boldsymbol{e}$ is not b, since the first clause of the formula would make the proposition true anyway. Thus, $\boldsymbol{e}$ can be written as $(\boldsymbol{y}bc \dots c)$ for a suitable word $\boldsymbol{y}$. We can partially compute the expression $\neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b})$ as follows.

$$\begin{aligned} \boldsymbol{a} &= (?\ 0\ 0 \dots 0) \\ \neg \boldsymbol{b} &= (?\ 0\ 1 \dots 1) \\ \boldsymbol{a} + \neg \boldsymbol{b} &= (?\ ?\ 1 \dots 1) \\ \neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b}) &= (?\ ?\ 1 \dots 1) \end{aligned}$$

On the other hand, suppose that the last bit of $\boldsymbol{b} \vee (\neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b}))$ is set to 1. Then, either the last bit of the vector $\boldsymbol{b}$ is 1, in which case $\boldsymbol{e}$ is certainly in L_0, or the last bit of $\neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b})$ is 1. We can thus assume that the last letter of $\boldsymbol{e}$ is c, and that the last bit of the binary sum $\boldsymbol{a} + \neg \boldsymbol{b}$ is 1, corresponding to the equation $0 + 1 = 1$. In the binary addition automaton,



we have $0 + 1 = 1$ if there was no occurrence of $1 + 1$ since the last occurrence of $0 + 0$, which literally translates as "*there was no occurrence of a since the last occurrence of b*". □

5 From Cascades to Bit-Vector Algorithms

At this stage, it remains only to put the pieces together. Corollary 1 and the Addition Lemma implies:

Theorem 2. *The output states of any counter-free automaton can be computed with a bit-vector algorithm.*

As an example, we will carry out the translation in the case of the automaton of Fig.1, giving a full justification of Formula (1). Fig. 2 gives a cascade decomposition of this automaton, that we present here in a slightly different – and more standard – way. In Fig. 4, the two copies of automaton $\mathcal{B}_1$ have been

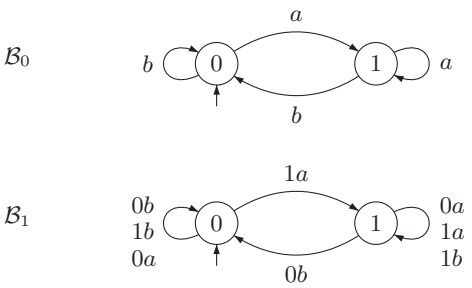


Fig. 4. A compact way to represent cascade decompositions

fused together, and transitions in $\mathcal{B}_1$ are prefixed by the label of the state of $\mathcal{B}_0$ to which they belong.

For example, in Fig. 4, the transition $0b$ represents the proposition: *automaton $\mathcal{B}_0$ was in state 0, and the current transition is b*. We already noted, in Section 3, that the automaton of Fig. 1 is in state 1 if and only if the cascade $\mathcal{B}_0 \circ \mathcal{B}_1$ is in global state $(0, 0)$. Using the Addition Lemma, automaton $\mathcal{B}_0$ is in state 0 if and only if:

$$\boldsymbol{b} \vee (\neg \boldsymbol{a} \wedge (\boldsymbol{a} + \neg \boldsymbol{b})), \tag{2}$$

and automaton $\mathcal{B}_1$ is in state 0 if and only if:

$$\mathbf{0}\boldsymbol{b} \vee (\neg \mathbf{1}\boldsymbol{a} \wedge (\mathbf{1}\boldsymbol{a} + \neg \mathbf{0}\boldsymbol{b})). \tag{3}$$

Since $\boldsymbol{b}$ implies $\neg \boldsymbol{a}$, the formula (2) reduces to $\boldsymbol{b}$. The formula (3) involves the two propositions:

$\mathbf{0}\boldsymbol{b}$: $\mathcal{B}_0$ was in state 0 and the current transition is b
$\mathbf{1}\boldsymbol{a}$: $\mathcal{B}_0$ was in state 1 and the current transition is a

which translate, respectively, as

1) $(\uparrow_1 \boldsymbol{b}) \wedge \boldsymbol{b}$
2) $\neg(\uparrow_1 \boldsymbol{b}) \wedge \boldsymbol{a}$.

Using the equivalence $\neg(\uparrow_1 \boldsymbol{b}) \Leftrightarrow (\uparrow_0 \neg \boldsymbol{b})$, the second proposition reduced to $\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}$. With this, Formula (3) becomes

$$(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b}) \vee (\neg(\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) \wedge ((\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) + \neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b}))).$$

Since $\boldsymbol{b}$ implies $\neg(\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a})$, the above formula is equivalent to

$$(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b}) \vee (\boldsymbol{b} \wedge ((\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) + \neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b}))). \tag{4}$$

Finally, using the logical equivalence $(p \wedge q) \vee (q \wedge r) \Leftrightarrow q \wedge (p \vee r)$, Formula(4) becomes

$$\boldsymbol{b} \wedge (\uparrow_1 \boldsymbol{b} \vee ((\uparrow_0 \boldsymbol{a} \wedge \boldsymbol{a}) + \neg(\uparrow_1 \boldsymbol{b} \wedge \boldsymbol{b}))$$

which is Formula (1).

5.1 Complexity Issues

The construction of the preceding section hides 'time-bombs' which are discussed in [4]. The first problem arises when some states of an automaton $\mathcal{A}$ must be encoded by exponentially many configurations of its cascade decomposition $\mathcal{B}_0 \circ \ldots \circ \mathcal{B}_{n-1}$. This implies that the length of the logical formulas that encode the languages recognized by those states can be exponential in the number of states of $\mathcal{A}$. Since the number of operations in the bit-vectors algorithms are proportional to the length of the logical formulas that code the languages, the negative results of [4] also apply to bit-vector algorithms.

Another potential pitfall of the method is that the Krohn-Rhodes Theorem does not provide efficient ways to obtain a decomposition. Moreover, deciding if an automaton is counter-free is NP-Hard [7].

Fortunately, computations that arise from biological problems involve automata that behave very well with respect to cascade decomposition. They belong to a particular class of automata for which it is possible to bound linearly – in the number of states and transitions – the size of the corresponding vector algorithm. We discuss this class in the next section.

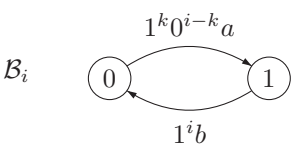
6 Solvable Automata Yield Nice Cascades

Throughout this section, we will suppose that $\mathcal{A}$ is a complete automaton with n states, and transition function δ. A *solvable* automaton [1] is an automaton for which there exists a labeling of its states from 1 to n such that, for any transition b:

$$\delta(k, b) < k \text{ implies } \forall k' \geq \delta(k, b), \quad \delta(k', b) = \delta(k, b).$$

If one thinks of the states of $\mathcal{A}$ as the output of a computation, the above property means that, if the output decreases, then its value depends only on the input, and not on the current state. Solvable automata appear, for instance, in algorithms used to compare biological sequences.

If automaton $\mathcal{A}$ is solvable, it admits a simple cascade decomposition of n binary reset automata $\mathcal{B}_0, \ldots, \mathcal{B}_{n-1}$. Each $\mathcal{B}_i$ is of the form:



with one reset transition $1^k0^{i-k}a$, to state 1, for each transition a and state k in automaton $\mathcal{A}$ such that:

$$\delta(k, a) > i \geq k,$$

and one reset transition 1^ib, to state 0, for each transition b and state k in automaton $\mathcal{A}$ such that:

$$\delta(k, b) \leq i < k.$$

Roughly, transitions of type a, that *increase* the value of the output state in $\mathcal{A}$, induce resets to state 1 in $\mathcal{B}_i$, and transitions of type b, that *decrease* the value of the output induce resets to state 0. Note that $\mathcal{B}_0$ has no resets.

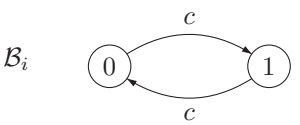
The following elementary properties of this construction can be easily checked:

Property 1 An increasing transition a defined in state k induces a reset to 1 in each automata $\mathcal{B}_k$ to $\mathcal{B}_{\delta(k,a)-1}$.

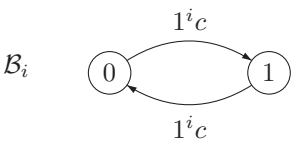
Property 2 A decreasing transition b to state $\delta(k, b)$ induces, by solvability, a reset to 0, labeled by $1^i b$, in each automata $\mathcal{B}_i$, for i from $\delta(k, b)$ to $n - 1$.

Lemma 2. *For each i, $0 \leq i \leq n-1$, $\mathcal{B}_i$ is a reset automaton.*

Proof. In order to show that $\mathcal{B}_i$ is a reset automaton, we have to show that there are no transition of the form



in any of the $\mathcal{B}_i$'s. By construction, such a transition could only be of the form



The transition $1^i c$ from state 0 to 1 implies that $\delta(i, c) > i$, and the transition $1^i c$ from state 1 to 0 implies $\exists j > i$ such that $\delta(j, c) \leq i < j$. Thus, $\delta(j, c) < j$ and solvability implies that

$$\forall k \geq \delta(j, c), \qquad \delta(k, c) = \delta(j, c).$$

Since $i \geq \delta(j, c)$, we must have $\delta(i, c) = \delta(j, c)$ which contradicts the hypothesis $\delta(i, c) > i$ and $\delta(j, c) \leq i$.

Lemma 3. *Let $\delta'(q_0 \dots q_{n-1}, \sigma) = (\delta_0(q_0, \sigma), \dots, \delta_{n-1}(q_0 \dots q_{n-2}, \sigma))$ denote the transition function of the cascade product $\mathcal{C} = \mathcal{B}_0 \circ \dots \circ \mathcal{B}_{n-1}$, then*

$$\delta(k, c) = j \text{ iff } \delta'(1^k 0^{n-k}, c) = 1^j 0^{n-j}.$$

Proof. We will consider three different cases:

1. $k < j$

   In this case, if $\delta(k, c) = j$, we have an increasing transition from state k and,

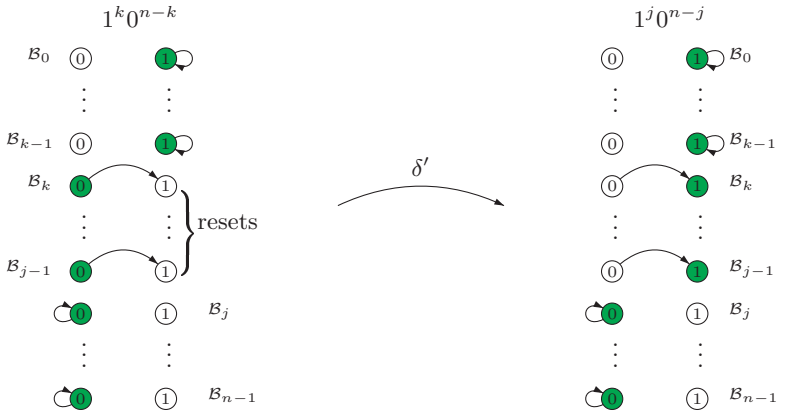


Fig. 5. Case $k < j$

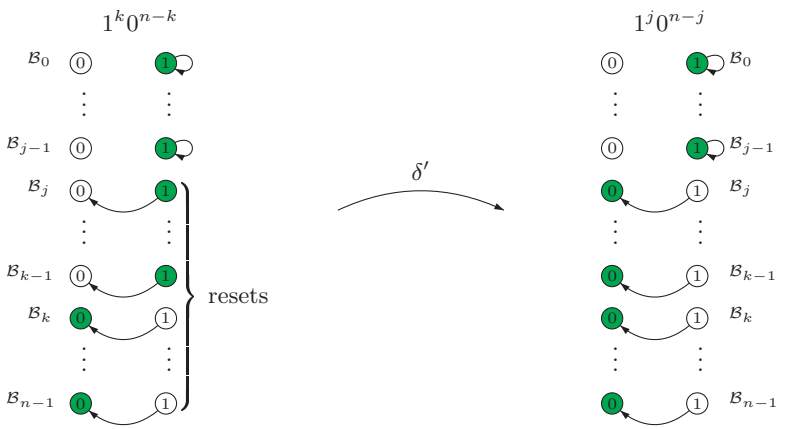


Fig. 6. Case $k > j$

by Property 1, c induces a reset to state 1 in each automata $\mathcal{B}_k$ to $\mathcal{B}_{j-1}$ in the cascade:

Thus $\delta'(1^k0^{n-k}, c) = 1^j0^{n-j}$. Conversely, if $\delta'(1^k0^{n-k}, c) = 1^j0^{n-j}$ then the resets of Fig. 5 are defined and, by Property 1, they can only be defined if $\delta(k, c) = j$.

2. $k > j$

   If $\delta(k, c) = j$, we have a decreasing transition from state k and, by Property 2, transition c induces a reset to state 0 in automata $\mathcal{B}_j$ to $\mathcal{B}_{n-1}$ in the cascade:
   Again we have that $\delta'(1^k0^{n-k}, c) = 1^j0^{n-j}$. Conversely, if $\delta'(1^k0^{n-k}, c) = 1^j0^{n-j}$, we have resets from state 1 to 0 in automata $\mathcal{B}_j$ through at least $\mathcal{B}_{k-1}$. And then, Property 2 implies that $\delta(k, c) = j$.

3. $j = k$

   In this case, if $\delta(k, c) = k$, transition c induces only identities in the cascade implying $\delta'(1^k0^{n-k}, c) = 1^k0^{n-k}$. Conversely, if $\delta'(1^k0^{n-k}, c) = 1^k0^{n-k}$, that means that for transition c no resets are defined in the cascade and we must have $\delta(k, c) = k$. (Every other possibilities would have induced resets in the cascade).

Using the above lemma, we can now state the basic result of this section:

Theorem 3. *The cascade $\mathcal{C} = \mathcal{B}_0 \circ \ldots \circ \mathcal{B}_{n-1}$ is a cascade decomposition of $\mathcal{A}$ with the homomorphism $\varphi(1^k0^{n-k}) = k$.*

Proof. Lemma 3 implies that the sub-automata of $\mathcal{C}$ generated by the set of states of the form $1^k0^{n-k}, k \geq 1$, is isomorphic to $\mathcal{A}$.

6.1 A Linear Bit-Vector Algorithm for Solvable Automata

The simple structure of the $\mathcal{B}_i$'s in the decomposition of the preceding section allows us to derive a linear algorithm for a solvable automaton $\mathcal{A}$ using Theorem 2. Consider the propositions:

$$\begin{aligned} P_i &: \text{ Automaton } \mathcal{A} \text{ goes to state } i. \\ Q_i &: \text{ Automaton } \mathcal{B}_i \text{ goes to state } 0. \end{aligned}$$

For i in $[1..n-1]$, Theorem 3 implies that P_i is equivalent to $Q_i \wedge \neg Q_{i-1}$, and P_n is simply $\neg Q_{n-1}$. Thus, knowing the values of the Q_i's, we can compute the output of $\mathcal{A}$ in $\mathcal{O}(n)$ steps.

For an automaton $\mathcal{B}_i$ in the cascade product, we first form the disjunction of all its resets, with the notations:

$$\begin{aligned} a_i &= \bigvee_{\delta(k,a)>i\geq k} 1^k0^{i-k}a \\ b_i &= \bigvee_{\delta(k,b)\leq i<k} 1^i b \end{aligned}$$

From the Addition Lemma, we immediately get the formula:

$$\boldsymbol{Q_i} = \boldsymbol{b_i} \vee (\neg \boldsymbol{a_i} \wedge (\boldsymbol{a_i} + \neg \boldsymbol{b_i})).$$

The tricky part is the fact that, since the disjonctions are over pairs of states and transitions, computing each of the vectors $\boldsymbol{a_i}$ and $\boldsymbol{b_i}$ can require $\mathcal{O}(mn)$ steps, where m is the size of the alphabet of $\mathcal{A}$. Thus, the total computing effort would be $\mathcal{O}(mn^2)$. Fortunately, there exists recursion formulas for both the $\boldsymbol{a_i}$'s and the $\boldsymbol{b_i}$'s, which reduce the total computing effort to $\mathcal{O}(mn)$. Here, I stands for the initial state of automaton $\mathcal{A}$.

Lemma 4. *The vectors $\boldsymbol{a_i}$ and $\boldsymbol{b_i}$, for $i \geq 1$ are given by:*

$$\boldsymbol{a_i} = (\neg \boldsymbol{Q'_{i-1}} \wedge (\bigvee_{\delta(i,a)>i} \boldsymbol{a})) \vee (\boldsymbol{a_{i-1}} \wedge \boldsymbol{Q'_{i-1}} \wedge \neg(\bigvee_{\delta(k,a)=i>k} (\uparrow_{i=k} \boldsymbol{P_k} \wedge \boldsymbol{a})))$$

$$\boldsymbol{b_i} = (\neg \boldsymbol{Q'_{i-1}} \wedge (\boldsymbol{b_{i-1}} \vee \bigvee_{\delta(k,b)=i<k} \boldsymbol{b}))$$

where $\boldsymbol{Q'_{i-1}} = \uparrow_{i>I} \boldsymbol{Q_{i-1}}$.

Proof: The formula for a_i can be split in two parts, according to whether $k = i$ or not:

$$a_i = \left(\bigvee_{\delta(i,a)>i} 1^i a \right) \vee \left(\bigvee_{\delta(k,a)>i>k} 1^k 0^{i-k} a \right).$$

Transitions of the form $1^i a$ mean that the preceding state of automaton $\mathcal{A}$ is at least i, thus the preceding state of $\mathcal{B}_{i-1}$ must be 1. Therefore, $\neg(\uparrow_{i>I} \boldsymbol{Q_{i-1}})$ must be true, where the boolean value $i > I$ takes care of the initial state. The first part of the disjunction thus becomes

$$(\neg \boldsymbol{Q'_{i-1}} \wedge (\bigvee_{\delta(i,a)>i} \boldsymbol{a})).$$

In the second part of the disjunction, transitions of the form $1^k 0^{i-k} a$, with $k < i$, mean that the preceding state is strictly less than i, which is equivalent to the formula $(\uparrow_{i>I} \boldsymbol{Q_{i-1}})$, and all transitions that were in $\boldsymbol{a_{i-1}}$ are in $\boldsymbol{a_i}$ except those for which $\delta(k, a) = i$. The formula for the $\boldsymbol{b_i}$'s is proved with similar arguments. □

Even if the formulas in Lemma 3 still seem to involve $\mathcal{O}(mn)$ steps, note that any increasing transition $\delta(k, a)$ of automaton $\mathcal{A}$ generates two terms, one in $\boldsymbol{a_k}$, and one in $\boldsymbol{a_{\delta(k,a)}}$. Any decreasing transition $\delta(k, b)$ generates only one term, in $\boldsymbol{b_{\delta(k,b)}}$. Thus, the overall computing effort is $\mathcal{O}(mn)$, even if computing some of the $\boldsymbol{a_i}$'s can also be $\mathcal{O}(mn)$.

7 Conclusions

We established that counter-free automata admit bit-vector algorithms, and that solvable automata admit linear bit-vector algorithms. Are the solvable automata the only ones that behave reasonably? One direction that was explored in this paper was to restrict the possible states of the cascade, and the type of resets allowable.

These restrictions characterize the class of solvable automata. Indeed, if one assumes that the states of the form $1^k 0^{n-k}$ are closed in a cascade of binary resets automata, then one can prove that the generated sub-automaton is solvable. The identification of other classes of automata that generate efficient vector algorithms should thus rely on a different approach.

References

[1] A. Bergeron and S. Hamel, *Vector Algorithms for Approximate String Matching* (to appear in IJFCS). 13, 21

[2] A. Bergeron and S. Hamel, *Cascade Decompositions are Bit-Vector Algorithms*, http://www.lacim.uqam.ca/~anne.

[3] K. Krohn and J. L. Rhodes, *Algebraic Theory of machines*, Transactions of the American Mathematical Society, 116, (1965), 450–464. 13

[4] O. Maler and A. Pnueli, *Tight Bounds on the Complexity of Cascaded Decomposition Theorem*, 31st Annual Symposium on Foundations of Computer Science IEEE, volume II, (1990), 672–682. 18, 21

[5] O. Maler and A. Pnueli, *On the Cascaded Decomposition of Automata, its Complexity and its Application to Logic*, unpublished manuscript available at http://www-verimag.imag.fr/PEOPLE/maler/uabst.html, (1994), 48 pages. 13

[6] E. Myers, *A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming* , J. ACM,46-3, (1999) 395–415. 13

[7] J. Stern, *Complexity of some Problems from the Theory of Automata*, Information and Control, 66, (1985), 163–176. 21

[8] H. P. Zeiger, *Cascade Synthesis of Finite-State Machines*, Information and Control, 10, (1967), 419–433. 15

Submodule Construction and Supervisory Control: A Generalization*

Gregor v. Bochmann

School of Information Technology and Engineering (SITE)
University of Ottawa, Canada
bochmann@site.uottawa.ca

Abstract. We consider the following problem: For a system consisting of two submodules, the behavior of one submodule is known as well as the desired behavior S of the global system. What should be the behavior of the second submodule such that the behavior of the composition of the two submodules conforms to S ? - This problem has also been called "equation solving", and in the context of supervisory control, it is the problem of designing a suitable controller (second submodule) which controls a given system to be controlled (first submodule). Solutions to this problem have been described by different authors for various assumptions about the underlying communication mechanisms and conformance relations. We present a generalization of this problem and its solution using concepts from relational database theory. We also show that several of the existing solutions are special cases of our general formulation

1 Introduction

In automata theory, the notion of constructing a product machine S from two given finite state machines S1 and S2, written S = S1 x S2, is a well-known concept. This notion is very important in practice since complex systems are usually constructed as a composition of smaller subsystems, and the behavior of the overall system is in many cases equal to the composition obtained by calculating the product of the behaviors of the two subsystems. Here we consider the inverse operation, also called equation solving: Given the composed system S and one of the components S1, what should be the behavior S2 of the second component such that the composition of these two components will exhibit a behavior equal to S. That is, we are looking for the value of X which is the solution to the equation S1 x X = S. This problem is an analogy of the integer division, which provides the solution to the equation N1 * X = N for integer values N1 and N. In integer arithmetic, there is in general no exact solution to this equation; therefore integer division provides the largest integer which multiplied with

* This work was partly supported by a research grant from the Natural Sciences and Engineering Research Council of Canada.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 27-39, 2002.
© Springer-Verlag Berlin Heidelberg 2002

N1 is smaller than N. Similarly, in the case of equation solving for machine composition, we are looking for the most general machine X which composed with S1 satisfies some conformance relation in respect to S. In the simplest case, this conformance relation is trace inclusion.

A first paper of 1980 [Boch 80d] (see also [Merl 83]) gives a solution to this problem for the case where the machine behavior is described in terms of labeled transition systems (LTS) which communicate with one another by synchronous interactions (see also [Hagh 99] for a more formal treatment). This work was later extended to the cases where the behavior of the machines is described in CCS or CSP [Parr 89], by finite state machines (FSM) communicating through message queues [Petr 98, Yevt 01a] or input/output automata [Qin 91, Dris 99], and to synchronous finite state machines [Kim 97].

The applications of this equation-solving method was first considered in the context of the design of communication protocols, where the components S1 and S2 may represent two protocol entities that communicate with one another [Merl 83]. Later it was recognized that this method could also be useful for the design of protocol converters in communication gateways [Kele 94, Tao 97a], and for the selection of test cases for testing a module in a context [Petr 96a]. It is expected that it could also be used in the other application domains where the re-use of components is important. If the specification of the desired system is given together with the specification of a module to be used as one component in the system, then equation solving provides the specification of a new component to be combined with the existing one.

Independently, the same problem was identified in control theory for discrete event systems [Rama 89] as the problem of finding a controller for a given system to be controlled. In this context, the specification S1 of the system to be controlled is given, as well as the specification of certain properties that the overall system, including the controller, should satisfy. If these properties are described by S, and the behavior of the controller is X, then we are looking for the behavior of X such that the equation S1 x X = S is satisfied. Solutions to this problem are described in [Bran 94] using a specification formalism of labeled transition systems where a distinction of input and output is made (interactions of the system to be controlled may be *controllable* (which corresponds to output of the controller) or *uncontrollable* (which correspond to input to the controller). This specification formalism seems to be equivalent to input/output automata (IOA).

In this paper we show that the above equation solving problem in the different contexts of LTS, communicating finite state machines (synchronous and asynchronous) and IOA are all special cases of a more general problem which can be formulated in the context of relational database theory which is a generalized to allows for non-finite relations (i.e. relations representing infinite sets). We also give the solution of this general problem. We show how the different specialized version of this problem - and the corresponding solutions - can be derived from the general database version.

These results were obtained after discussions with N. Yevtushenko about the similarity of the formulas that describe the solution of the equation in [Yevt 01a] and [Merl 80]. The generalization described here became apparent after listening to a talk on stochastic relational databases by Cory Butz. In fact, it appears that the solution in

the context of relational databases, as described in this paper, can be extended to the case of Bayesian databases.

2 Review of Some Notions from the Theory of Relational Databases

The following concepts are defined in the context of the theory of relational databases [Maie 83]. Informally, a relational database is a collection of relations where each relation is usually represented as a table with a certain number of columns. Each column corresponds to an attribute of the relation and each row of the table is called a tuplet. Each tuplet defines a value for each attribute of the relation. Such a tuplet represents usually an "object", for instance, if the attributes of the *employee* relation are *name*, *city*, *age*, then the tuplet <Alice, Ottawa, 25> represents the employee "Alice" from "Ottawa" who is 25 years old.

The same attribute may be part of several relations. Therefore we start out with the definition of all attributes that are of relevance to the system we want to describe.

Definition ***(attributes and their values)****:* The set $A = \{a_1, a_2, \ldots, a_m\}$ is the set of attributes. To each attribute a_i is associated a (possibly infinite) set D_i of possible values that this attribute may take. D_i is called the domain of the attribute a_i . We define $D = \cup D_i$ to be the discriminate union of the D_i .

Definition ***(relation)****:* Given a subset A_r of A, a relation R over A_r, written $R[A_r]$, is a (possibly infinite) set of mappings $T: A_r \dashrightarrow D$ with $T(a_i) \in D_i$. An **integrity constraint** is a predicate on such mappings. If the relation R has an integrity constraint C, this means that for each $T \in R$, C(T) is true.

Note: In the informal model where a relation is represented by a table, a mapping T corresponds to a tuplet in the table. Here we consider relations that may include an infinite number of different mappings.

Definition ***(projection)****:* Given $R[A_r]$ and $A_x \subseteq A_r$, the projection of $R[A_r]$ onto A_x , written $proj_{Ax}$ (R), is a relation over A_x with

$T \in proj_{Ax}(R)$ iff there exists $T' \in R$ such that for all $a_i \in A_x$, $T(a_i) = T'(a_i)$

We note that here T is the restriction of T' to the subdomain A_x . We also write $T = proj_{Ax}(T')$.

Definition ***(natural join)****:* Given $R_1[A_1]$ and $R_2[A_2]$, we define the (natural) join of the relations R_1 and R_2 to be a relation over $A_1 \cup A_2$, written R_1 join R_2 , with

$T \in (R_1 \text{ join } R_2)$ iff $proj_{A1}(T) \in R_1$ and $proj_{A2}(T) \in R_2$

Definition ***(chaos)****:* Given $A_r \subseteq A$, we call chaos over A_r , written $Ch[A_r]$, the relation which includes all elements T of $A_r \dashrightarrow D$ with $T(a_i) \in D_i$, that is, the union of all relations over A_r.

Note: We note that $Ch[A_r]$ is the Cartesian product of the domains of all the attributes in A_r . The notion of "chaos" is not common in database theory. It was introduced by Hoare [Hoar 85] to denote the most general possible behavior of a module. It was also used in several papers on submodule construction [xxFSM, Dris 99b].

It is important to note that we consider here infinite attribute value domains and relations that contain an infinite number of mappings (tuplets). In the context of traditional database theory, these sets are usually finite (although some results on infinite databases can be found in [Abit 95]). This does not change the form of our definitions, however. If one wants to define algorithms for solving equations involving such infinite relations, one has to worry about the question of what kind of finite representations should be adopted to represent these relations. The choice of such representations will determine the available algorithms and at the same time introduce restrictions on the generality of these algorithms. Some of these representation choices are considered in Sections 4 and 5.

3 Equation Solving in the Context of Relational Databases

3.1 Some Interesting Problems (Simplest Configuration)

In the simple configuration assumed in this subsection, we consider three attributes a_1, a_2, and a_3, and three relations $R_1[\{a_2, a_3\}]$, $R_2[\{a_1, a_3\}]$, and $R_3[\{a_2, a_1\}]$. Their relationship is informally shown in Figure 3.1.

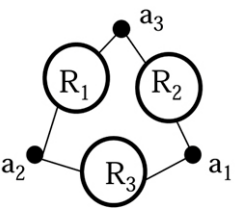


Fig. 3.1. Configuration of 3 relations sharing 3 attributes

We consider the following equation (which is in fact an inclusion relation)

$$\text{proj}_{\{a2,\,a1\}}\,(R_1 \text{ join } R_2) \subseteq R_3 \qquad (1)$$

If the relations R_1 and R_3 are given, we can ask the question: for what relation R_2 will the above equation be true. Clearly, the empty relation, $R_2 = \Phi$ (empty set), satisfies this equation. However, this case is not very interesting. Therefore we ask the following more interesting questions for the given relations R_1 and R_3 :

Problem (1): Is there a maximal relation R_2 that satisfies the above equation (maximal in the sense of set inclusion; any larger relation is no solution) ?

Problem (2): Could there be more than one maximal solution (clearly not including one another) ?

Problem (3): Is there a solution for the case when the $\subseteq$ operator is replace by equality of by the $\supseteq$ operator ?

3.2 Some Solutions

First we note that there is always a single maximal solution. This solution is the set

$$Sol^{(2)} = \{T \varepsilon Ch[\{a_1, a_3\}] \mid proj_{\{a2, a1\}} (R_1 \text{ join } \{T\}) \subseteq R_3 \} \quad (2)$$

This is true because the operators of set union and intersection obey the distributive law in respect to the projection and join operations, that is, $proj_{Ax} (R_i \text{ union } R_j) = proj_{Ax} (R_i) \cup proj_{Ax} (R_j)$; and similarly for intersection and the join operations.

While the above characterization of the solution is trivial, the following formula is useful for deriving algorithms that obtain the solution in the context of the specific representations discussed in Sections 4 and 5.

Theorem: A solution for R_2 that satisfies Equation (1), given R_1 and R_3 , is given by the following formula (where "/" denotes set substraction):

$$Sol^{(3)} = Ch[\{a_1, a_3\}] / proj_{\{a1, a3\}} (R_1 \text{ join } (Ch[\{a_1, a_2\}] / R_3)) \quad (3)$$

This is the largest solution and all other solutions of Equ. (1) are included in this one.

Informally, Equation (3) means that the largest solution consists of all tuplets over $\{a_1, a_3\}$ that cannot be obtained from a projection of a tuplet T $[\{a_1, a_2, a_3\}]$ that can be obtained by a join from an element of R_1 and a tuplet from $Ch[\{a_1, a_2\}]$ that is not in R_3. A formal proof of this theorem is given in [Boch 01b].

We note that the smaller solution

$$Sol^{(3*)} = proj_{\{a1, a3\}} (R_1 \text{ join } R_3) / proj_{\{a1, a3\}} (R_1 \text{ join } (Ch[\{a_1, a_2\}] / R_3)) \quad (3*)$$

is also an interesting one, because it contains exactly those tuplets of $Sol^{(3)}$ that can be joint with some tuplet of R_1 to result in a tuplet whose projection on $\{a_1, a_2\}$ is in R_3 . Therefore (R_1 join $Sol^{(3)}$) and (R_1 join $Sol^{(3*)}$) are the same set of tuplets; that means the same subset of R_3 is obtained by these two solutions. In this sense, these solutions are equivalent. We note that the solution formula given in [Merl 83] corresponds to the solution $Sol^{(3*)}$.

3.3 Some Simple Example

We consider here a very simple example of three relations $R_1[\{a_2, a_3\}]$, $R_2[\{a_1, a_3\}]$, and $R_3[\{a_2, a_1\}]$ as discussed above and shown in Figure 3.1. We assume that the domains of the attributes are as follow:
$D_1 = \{n\}$, $D_2 = \{aa, ab, ba, bb\}$ and $D_3 = \{c, d\}$. We assume that R_1 and R_3 contain the tuplets shown in Figure 3.2 below. Then the evaluation of the solution formula Equation (3) leads to some intermediate results and the solution $Sol^{(3)}$, also shown in the figure.

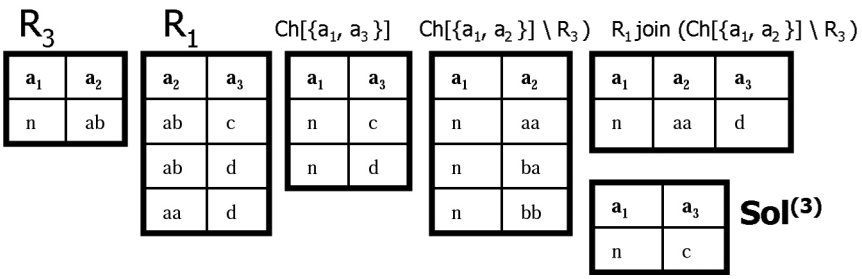


Fig. 3.2. Example of database equation solving (Example 1)

3.4 A more General Setting of the Problem

In Section 3.2 we assumed that all the three relations have two attributes and that each pair of relations share exactly one attribute. However, we may consider more general situations, such as shown in Figure 3.3. Here we consider different subsets A_1, A_2, A_3 and A_0 of the global set of attributes A. The subsets A_1, A_2, and A_3 correspond to the attributes a_1, a_2, and a_3 considered in Section 3.1, while the subset A_0 is a set of attributes that are shared by all three relations.

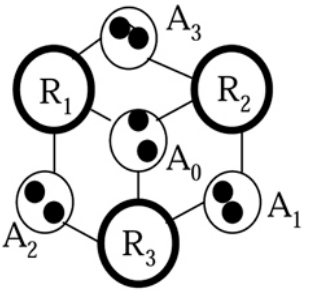


Fig. 3.3. Configuration of 3 relations sharing various attributes

The generalization of Equation (1) is then defined as follows. We consider the three relations $R_1[A_2 \cup A_3 \cup A_0]$, $R_2[A_1 \cup A_3 \cup A_0]$, , and $R_3[A_2 \cup A_1 \cup A_0]$. We consider the equation

$$\text{proj}_{(A2 \cup A1 \cup A0)} (R_1 \text{ join } R_2) \subseteq R_3 \qquad (1')$$

If the relations R_1 and R_3 are given, the largest relation R_2 that satisfies the above equation is then characterized by the formula

$$Sol^{(3')} = Ch[A_1 \cup A_3 \cup A_0] \ / \text{proj}_{(A1 \cup A3 \cup A0)} (R_1 \text{ join } (Ch[A_1 \cup A_2 \cup A_0] \ / R_3)) \qquad (3')$$

The proof of this equation is similar to the proof of Equation (3).

4 Equation Solving in the Context of Composition of Sequential Machines or Reactive Software Components

4.1 Modeling System Components and Behavior Using Traces

Sequential machines and reactive software components are often represented as black boxes with *ports*, as shown in Figure 4.1. The ports, shown as lines in Figure 4.1, are the places where the *interactions* between the component in question and the components in its environment take place. Sometimes arrows indicate the direction of the interactions, implying that one component produces the interaction as output while the other component(s) accept it as input. This distinction is further discussed in Section 5.

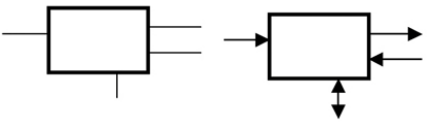


Fig. 4.1. Components and their ports

For allowing the different modules to communicate with one another, their ports must be interconnected. Such interconnection points are usually called *interfaces*. An example of a composition of three modules (sequential machines or reactive software components) is shown in Figure 4.2. Their ports are pair-wise interconnected at three interfaces a_1, a_2, and a_3.

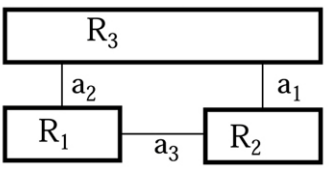


Fig. 4.2. Configuration of 3 components interconnected through 3 interfaces

The dynamic behavior of a module (sequential machine or a reactive software component) is usually described in terms of *traces*, that is, sequences of interactions that take place at the interfaces to which the module is connected. Given an interconnection structure of several modules and interfaces, we define for each interface i the set of possible interactions I_i that may occur at that interface. For each (finite) system execution trace, the sequence of interactions observed at the interface a_i is therefore an element of I_i * (a finite sequence of elements in I_i).

For communication between several modules, we consider in this paper rendezvous interactions. This means that, for an interaction to occur at an interface, it is necessary that all modules connected to that interface must make a state transition compatible with that interaction at that interface.

In our basic communication model we assume that the interactions between the different modules within the system are synchronized by a clock, and that there must be an interaction at each interface during each clock period. We call this "synchronous operation".

4.2 Correspondence with the Relational Database Model

We note that the above model of communicating system components can be described in the formalism of (infinite) relational databases as follows:

1. A port corresponds to an attribute and a module to a relation. For instance, the interconnection structure of Figure 4.2 corresponds to the relationship shown in Figure 3.1. The interfaces a_1, a_2, and a_3 in Figure 4.2 correspond to the three attributes a_1, a_2, and a_3 introduced in Section 2.2, and the three modules correspond to the three relations.

2. If a given port (or interface) corresponds to a particular attribute a_i, then the possible execution sequences I_i^* occurring at that port correspond to the possible values of that interface, i.e. $D_i = I_i^*$.
3. The behavior of a module M_x is given by the tuplets T_x contained in the corresponding relation R_x [A_x], where A_x corresponds to the set of ports of M_x. That is, a trace t_x of the module X corresponds to a tuplet T_x which assigns to each interface a_i the sequence of interactions s_{xi} observed at that interface during the execution of this trace. We write $s_{xi}{}^{@t}$ to denote the t-th element of s_{xi}

Since we assume "synchronous operation" (as defined in Section 4.1), all tuplets in a relation describing the behavior of a module must satisfy the following constraint:

Synchrony Constraint: The length of all attribute values are equal. (This is the length of the trace described by this tuplet.)

In many cases, one assumes that the possible traces of a module are closed under the prefix relation, however, this is not necessary for the following discussion. In this case then, a relation R[A] describing the behavior of a module must also satisfy the following constraint:

Prefix-closure Constraint: If $T_x \varepsilon R$ and T_y is such that s_{xi} is a prefix of s_{yi} for all $i \varepsilon A$ (and T_y satisfies the synchrony constraint), then $T_y \varepsilon R$.

As an example we consider two module behaviors R_1 and R_3 which have some similarity with the relations R_1 and R_3 considered in the database example of Section 3.3. These behaviors are described in the form of finite state transition machines in Figure 4.3. The interactions at the interface a_2 are a, b or n, the interactions at a_3 are c, d or n, and the interface a_1 only allows the interaction n. The notation b/n for some state transition means that this transition occurs when at one interface the interaction b occurs and at the other interface the interaction n. For instance, the traces of length 3 defined by the behavior of R_1 are (a/n, n/c. b/n), (a/n, n/d, b/n), and (a/n, n/d, a/n), which are similar, in some sense, to the tuplets in the relation R_1 of the example in Section 3.3.

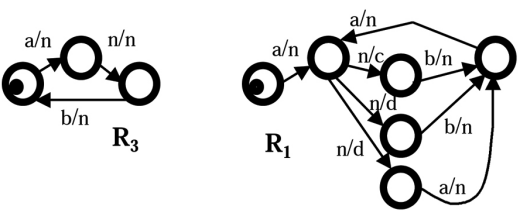


Fig. 4.3. Behavior specifications R_1 and R_3 (Example 2)

4.3 The Case of Synchronous Finite State Machines

If we restrict ourselves to the case of regular behavior specifications, where the (infinite) set of traces of a module can be described by a finite state transition model, we can use Equation (3) or Equation (3*) to derive an algorithm for equation solving. We note that the algorithm reported in [Yevt 01a] corresponds to Equation (3). Similar work is also described in [Kim 97] and [Qin 91]. In this case, the behavior specifica-

tion for a module is given in the form of a finite state transition diagram where each transition is labeled by a set of interactions, one for each port of the module, as in the example above.

The algorithm for equation solving is obtained from Equation (3) or Equation (3*) by replacing the relational database operators *projection*, *join* and *substraction* by the corresponding operations on finite state automata. The database *projection* corresponds to eliminating those interaction labels from all transitions of the automaton which correspond to attributes that are not included in the set of ports onto which the projection is done. This operation, in general, introduces nondeterminism in the resulting automaton. The *join* operation corresponds to the composition operator of automata which is of polynomial complexity (see above references for more details). The *substraction* operation is of linear complexity if its two arguments are deterministic. Since the projection operator introduces nondeterminism, one has to include a step to transform the automata into their equivalent deterministic forms. This step is of exponential complexity. Therefore the equation solving algorithm for synchronous finite state machines is of exponential complexity. However, our experience with some examples involving the interleaved semantics described below [Dris 99a] indicates that reasonably complex systems can be handled in many cases.

4.4 The Case of Interleaving Rendezvous Communication

Under this subsection, we consider non-synchronous rendezvous communication also called interleaving semantics, were at each instant in time at most one interaction takes place within all interconnected system components. This communication paradigm is used for instance with labeled transition systems (LTS). One way to model the behavior of such systems is to consider a global execution trace which is the sequence of interactions in the order in which they take place at the different interfaces (one interface at a time). Each element of such an execution sequence defines the interface a_i at which the interaction occurred and the interaction v_i which occurred at this interface.

Another way to represent the behavior of such systems is to reduce it to the case of synchronous communication as follows. This is the approach which we adopt in this paper because it simplifies the correspondence with the relational database model. In order to model the interleaving semantics, we postulate that all sets I_i include a dummy interaction, called *null*. It represents the fact that no interaction takes place at the interface. We then postulate that each tuplet T of a relation R[A] satisfies the following constraint:

Interleaving Constraint: For all time instants t ($t > 0$) we have that $T(a_i)[t] \neq null$ implies $T(a_j)[t] = null$ for all $a_j \in A$ ($j \neq i$).

We note that tuplets that are equal to one another except for the insertion of time periods during which all interfaces have the null interaction are equivalent (called stuttering equivalence). One may adopt a normal form representation for such an equivalence class in the form of the execution sequence (in this class) that has no time instance with only null interactions. This execution sequence is trivially isomorphic to the corresponding interaction sequence in the first interleaving model considered above.

We note that we may assume that all relations satisfy the constraint that they are closed under stuttering, that is, T ε R implies that R also contains all other tuplets T' that are stuttering equivalent to T.

4.5 The Case of Finite Labeled Transition Systems

The interleaving rendezvous communication is adopted for labeled transition systems (LTS) (voir e.g. [Hoare 85]). To simplify the notation, we assume that the sets of interactions at different interfaces are disjoint (i.e. I_i intersection I_j = empty for $a_i \neq a_j$), and we introduce the overall set of interactions $I = U_{(ai\ \varepsilon\ A)}\ I_i$. Then a class of stuttering equivalent interleaving traces (as described in Section 4.4) correspond one-to-one to a sequence of interactions in I.

If we restrict ourselves to the case where the possible traces of a module are described by a finite LTS, the resulting set of possible execution sequences are regular sets and the operations *projection*, *join* and *substraction* over interleaving traces can be represented by finite operations over the corresponding LTS representations. The situation is similar as in the case of synchronous finite state machines, discussed in Section 4.3, because of the nondeterminism introduced by the project operator, the substraction operation becomes of exponential complexity. The projection operation corresponds to replacing the interaction labels of transitions that correspond to ports that are not included in the projected set by a spontaneous transition label (sometimes written "i"). The join operation is the standard LTS composition operation, and the determination and substraction operations can be found in standard text books of automata theory.

As an example, we may consider the behavior specifications given in Figure 4.3. If we interpret the interaction "n" as the null interaction, then the behaviors R_1 and R_3 satisfy the interleaving constraint described above and can be interpreted as labeled transition systems. Their traces can be characterized by the regular expressions " (a . b)* " and " (a . (c . b + d . b + d . a))* ", respectively. If we execute the algorithm implied by Equation (3) we obtain the solution behavior for R_2 which can be characterized by "c*". This solution is similar to the solution for the database example discussed in Section 3.3.

5 Conclusions

The problem of submodule construction (or equation solving for module composition) has some important applications for the real-time control systems, communication gateway design, and component re-use for system design in general. Several algorithms for solving this problem have been developed based on particular formalisms that were used for defining the dynamic behavior of the desired system and the existing submodule. In this paper, we have shown that this problem can also be formulated in the context of relational databases. The solution to the problem is given in the form of a set-theoretical formula which defines the largest relation that is a solution of the equation.

Whether this solution is useful for practical applications in the context of relational databases is not clear. However, we have shown here that the formulation of this

problem in the context of relational databases is a generalization of several of the earlier approaches to submodule construction, in particular in the context of synchronous finite state machines [Kim 97, Yevt 01a], Labelled Transition Systems (LTS) [Merl 83]. In the case of regular behavior specifications in the form of finite transition machines, the set-theoretical solution formula of the database context can be used to derive solution algorithms based on the finite representations of the module behaviors, which correspond to those described in the literature.

In [Boch 01b], the submodule construction problem is addressed for the case of synchronous communication with the distinction of input and output (implying a module specification paradigm with hypothesis and guarantees. A general solution formula in the spirit of Equation (3) is given for this case. This solution can be used to derive solution algorithms for the case of synchronous finite state machines with input/output distinction and of Input/Output Automata (as described in [Dris 99c] and finite state machines communicating through message queues (as described in [Petr 98].

We believe that these solution formulas can also be used to derive submodule construction algorithms for specification formalism that consider finer conformance relations than simple trace semantics (as considered in this paper). Examples of existing algorithms of this class are described in [This 95] for considering liveness properties and in [Bran 94, Male95, Dris 00] for considering hard real-time properties. Some other work [Parr 89] was done in the context of the specification formalism CSP [Hoare 85] and observational equivalence for which it is known that no solution algorithm exists because the problem is undecidable.

Acknowledgements

I would like to thank the late Philip Merlin with whom I started to work in the area of submodule construction. I would also like to thank Nina Yevtushenko (Tomsk University, Russia) for many discussions about submodule construction algorithms and the idea that a generalization of the concept could be found for different behavior specification formalisms. I would also like to thank my former colleague Cory Butz for giving a very clear presentation on Bayesian databases which inspired me the database generalization described in Section 3 in this paper. Finally, I would like to thank my former PhD students Z.P. Tao and Jawad Drissi whose work contributed to my understanding of this problem.

References

[Abit 95] S. Abiteboul, R. Hull and V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[Boch 80d] G. v. Bochmann and P. M. Merlin, On the construction of communication protocols, ICCC, 1980, pp.371-378, reprinted in "Communication Protocol Modeling", edited by C. Sunshine, Artech House Publ., 1981; russian translation: Problems of Intern. Center for Science and Techn. Information, Moscow, 1981, no. 2, pp. 146-155.

[Boch 01b] G. v. Bochmann, Submodule construction - the inverse of composition, Technical Report, Sept. 2001, University of Ottawa.

[Bran 94] B. A. Brandin and W. M. Wonham, Supervisory Control of Timed Discrete-Event Systems, IEEE Tran. on Automatic Control, Vol.39, No.2, Feb. 1994.

[Dris 99a] J. Drissi and G. v. Bochmann, Submodule construction tool, in Proc. Int. Conf. on Computational Intelligence for Modelling, Control and Automation, Vienne, Febr. 1999, (M. Mohammadian, Ed.), IOS Press, pp. 319-324.

[Dris 99b] J. Drissi and G. v. Bochmann, Submodule construction for systems of I/O automata, submitted for publication.

[Dris 00] J. Drissi and G. v. Bochmann, Submodule construction for systems of timed I/O automata, submitted for publication, see also J. Drissi, PhD thesis, University of Montreal, March 2000 (in French).

[Hagh 99] E. Haghverdi and H. Ural, Submodule construction from concurrent system specifications, Information and Software Technology, Vo. 41 (1999), pp. 499-506.

[Hoar 85] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[Kele 94] S. G. H. Kelekar, Synthesis of protocols and protocol converters using the submodule construction approach, Proc. PSTV, XIII, A. Danthine et al (Eds), 1994.

[Kim 97] T.Kim, T.Villa, R.Brayton, A.Sangiovanni-Vincentelli. Synthesis of FSMs: functional optimization. Kluwer Academic Publishers, 1997.

[Maie 83] D. Maier, The Theory of Relational Databases, Computer Science Press, Rockville, Maryland, 1983.

[Male 95] O. Maler, A. Pnueli and J. Sifakis, On the synthesis of discrete controllers for timed systems, STACS 95, Annual Symp. on Theoretical Aspects of Computer Science, Berlin, 1995, Springer Verlag, pp. 229-242.

[Merl 83] P. Merlin and G. v. Bochmann, On the Construction of Submodule Specifications and Communication Protocols, ACM Trans. on Programming Languages and Systems, Vol. 5, No. 1 (Jan. 1983), pp. 1-25.

[Parr 89] J. Parrow, Submodule Construction as Equation Solving in CCS, Theoretical Computer Science, Vol. 68, 1989.

[Petr 96a] A. Petrenko, N. Yevtushenko, G. v. Bochmann and R. Dssouli, Testing in context: framework and test derivation, Computer Communications Journal, Special issue on Protocol engineering, Vol. 19, 1996, pp.1236-1249.

[Petr 98] A. Petrenko and N. Yevtushenko, Solving asynchronous equations, in Proc. of IFIP FORTE/PSTV'98 Conf., Paris, Chapman-Hall, 1998.

[Qin 91] H. Qin and P. Lewis, Factorisation of finite state machines under strong and observational equivalences, J. of Formal Aspects of Computing, Vol. 3, pp. 284-307, 1991.

[Rama 89] P. J. G. Ramadge and W. M. Wonham, The control of discrete event systems, in Proceedings of the IEEE, Vo. 77, No. 1 (Jan. 1989).

[Tao 97a] Z. Tao, G. v. Bochmann and R. Dssouli, A formal method for synthesizing optimized protocol converters and its application to mobile data networks, Mobile Networks & Applications, vol. 2, no. 3, 1997, pp. 259-69. Publisher: Baltzer; ACM Press, Netherlands.

[Tao 95d] Z. P. Tao, G. v. Bochmann and R. Dssouli, A model and an algorithm of subsystem construction, in proceedings of the Eighth International Conference on parallel and distributed computing systems, Sept. 21-23, 1995 Orlando, Florida, USA, pp. 619-622.

[This 95] J. G. Thistle, On control of systems modelled as deterministic Rabin automata, Discrete Event Dynamic Systems: Theory and Applications, Vol. 5, No. 4 (Sept. 1995), pp. 357-381.

[Yevt 01a] N. Yevtushenko, T. Villa, R. Brayon, A. Petrenko, A. Sangiovanni-Vincentelli. Synthesis by language equation solving (exended abstract), in Proc. of Annual Intern.workshop on Logic Snthesis, 2000, 11-14; complete paper to be published in ICCAD'2001; see also Solving Equations in Logic Synthesis, Technical Report, Tomsk State University, Tomsk, 1999, 27 p. (in Russian).

Counting the Solutions of Presburger Equations without Enumerating Them*

Bernard Boigelot and Louis Latour

Institut Montefiore, B28, Université de Liège
B-4000 Liège Sart-Tilman, Belgium
{boigelot,latour}@montefiore.ulg.ac.be
http://www.montefiore.ulg.ac.be/~{boigelot,latour}

Abstract. The *Number Decision Diagram (NDD)* has recently been proposed as a powerful representation system for sets of integer vectors. In particular, NDDs can be used for representing the sets of solutions of arbitrary Presburger formulas, or the set of reachable states of some systems using unbounded integer variables. In this paper, we address the problem of counting the number of distinct elements in a set of vectors represented as an NDD. We give an algorithm that is able to perform an exact count without enumerating explicitly the vectors, which makes it capable of handling very large sets. As an auxiliary result, we also develop an efficient projection method that allows to construct efficiently NDDs from quantified formulas, and thus makes it possible to apply our counting technique to sets specified by formulas. Our algorithms have been implemented in the verification tool LASH, and applied successfully to various counting problems.

1 Introduction

Presburger arithmetic [Pre29], i.e., the first-order additive theory of integers, is a powerful formalism for solving problems that involve integer variables. The manipulation of sets defined in Presburger arithmetic is central to many kinds of applications, including integer programming problems [Sch86, PR96], compiler optimization techniques [Pug92], temporal database queries [KSW95], and program analysis tools [FO97, SKR98].

The more direct way of handling algorithmically Presburger-definable sets consists of using a formula-based representation system. This approach has been successfully implemented in the *Omega package* [Pug92], which is probably the most widely used Presburger tool at the present time. Unfortunately, formula-based representations suffer from a serious drawback : They lack canonicity, which implies that sets with a simple structure are in some situations represented by very complex formulas; this notably happens when these formulas are

* This work was partially funded by a grant of the "Communauté française de Belgique — Direction de la recherche scientifique — Actions de recherche concertées", and by the European Commission (FET project ADVANCE, contract No IST-1999-29082).

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 40–51, 2002.
© Springer-Verlag Berlin Heidelberg 2002

obtained as the result of lengthy sequences of operations. Moreover, the absence of a canonical representation hinders the efficient implementation of usually essential decision procedures, such as testing whether two sets are equal to each other.

In order to alleviate these problems, an alternative representation of Presburger-definable sets has been developed, based on finite-state automata. The *Number Decision Diagram (NDD)* [WB95, Boi99] is, sketchily, a finite-state machine recognizing the encodings of the integer vectors belonging to the set that it represents. Its main advantage are that most of the usual set-theory operations can be performed by simply carrying out the corresponding task on the languages accepted by the automata, and that a canonical representation of a set can easily be obtained by minimizing its associated automaton. Among its applications, the NDD has made it possible to develop a tool for computing automatically the set of reachable states of programs using unbounded integer variables [LASH].

The problem of counting how many elements belong to a Presburger-definable set has been solved for formula-based representations [Pug94] of Presburger sets. Though of broad scope, this problem has interesting applications related to verification and program analysis. First, it can be used in order to quantify precisely the performances of some systems. In particular, one can estimate the computation time of code fragments or the amount of resources that they consume wherever these quantities can be expressed as Presburger formulas. Furthermore, counting the number of reachable data values at some control locations makes it possible to detect quickly some inconsistencies between different releases of a program, without requiring to write down explicit properties. For instance, it can promptly alert the developer, although without any guarantee of always catching such errors, that a local modification had an unwanted influence on some remote part of the program. Finally, studying the evolution of the number of reachable states with respect to the value of system parameters can also help to detect unsuspected errors.

The main goal of this paper is to present a method for counting the number of elements belonging to a Presburger-definable set represented by an NDD. Intuitively, our approach is based on the idea that one can easily compute the number of distinct paths of a directed acyclic graph without enumerating them. The actual algorithm is however more intricate, due to the fact that the vectors belonging to a set and the accepting paths of its representing NDD are not linked to each other by a one-to-one relationship.

In order to apply our counting technique to the set of solutions of a given Presburger formula, one needs first to build an NDD from that formula. This problem has been solved in [BC96, Boi99], but only in the form of a construction algorithm that is exponentially costly in the number of variables involved in the formula. As an auxiliary contribution of this paper, we describe an improved algorithm for handling the problematic projection operation. The resulting construction procedure has been implemented and successfully applied to problems involving large numbers of variables.

2 Basic Notions

We here explain how finite-state machines can represent sets of integer vectors. The main idea consists of establishing a mapping between vectors and words. Our encoding scheme for vectors is based on the classical expression of numbers in a base $r > 1$, according to which an encoding of a positive integer z is a word $a_{p-1}a_{p-2}\cdots a_1a_0$ such that each *digit* a_i belongs to the finite alphabet $\{0, 1, \ldots, r-1\}$ and $z = \sum_{i=0}^{p-1} a_i r^i$. Negative numbers z have the same p-digit encoding as their r's complement $r^p + z$. The number p of digits is not fixed, but must be large enough for the condition $-r^{p-1} \leq z < r^{p-1}$ to hold. As a corollary, the first digit of the encodings is 0 for positive numbers and $r-1$ for negative ones, hence that digit is referred to as the *sign digit* of the encodings.

In order to encode a vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)$, one simply reads repeatedly and in turn one digit from the encodings of all its components, under the additional restriction that these encodings must share the same length. In other words, an encoding of $\boldsymbol{v}$ is a word $d_{p-1,1}d_{p-1,2}\ldots d_{p-1,n}d_{p-2,1}d_{p-2,2}\ldots d_{0,n-1}d_{0,n}$ such that for every $i \in \{1, \ldots, n\}$, $d_{p-1,i}d_{p-2,i}\ldots d_{0,i}$ is an encoding of v_i. An encoding of a vector of dimension n has thus n sign digits — each associated to one vector component — the group of which forms a *sign header*.

Let $S \subseteq \mathbf{Z}^n$ be a set of integer vectors. If the language $L(S)$ containing all the encodings of all the vectors in S is regular, then any finite-state automaton accepting $L(S)$ is a *Number Decision Diagram (NDD)* representing S. It is worth noticing that, according to this definition, not all automata defined over the alphabet $\{0, 1, \ldots, r-1\}$ are valid NDDs. Indeed, an NDD must accept only valid encodings of vectors that share the same dimension, and must accept all the encodings of the vectors that it recognizes. Note that the vector encoding scheme that we use here is slightly different from the one proposed in [BHMV94, Boi99], in which the digits related to all the vector components are read simultaneously rather than successively. It is easy to see that both representation methods are equivalent from the theoretical point of view, the advantage of our present choice being that it produces considerably more compact finite-state representations. For instance, a minimal NDD representing $\mathbf{Z}^n$ is of size $O(2^n)$ if it reads component digits simultaneously, which limits the practical use of that approach to small values of n. On the other hand, our improved encoding scheme yields an automaton of size $O(n)$.

It is known for a long time [Cob69, Sem77] that the sets that can be represented by finite-state automata in every base $r > 1$ are exactly those that are definable in *Presburger arithmetic*, i.e., the first-order theory $\langle \mathbf{Z}, +, \leq \rangle$. One direction of the proof of this result is constructive, and translates into a algorithm for constructing an NDD representing an arbitrary Presburger formula [BHMV94]. Sketchily, the idea is to start from elementary NDDs corresponding to the formula atoms, and to combine them by means of set operators and quantifiers. It is easily shown that computing the union, intersection, difference or Cartesian product of two sets represented by NDDs is equivalent to carrying out similar operations on the languages accepted by the underlying automata. Quantifying existentially a set with respect to a vector component, which amounts to *pro-*

jecting this set along this component, is more tedious. We discuss this problem in the next section.

At this time, one could wonder why we did not opt for defining NDDs as automata accepting only one encoding (for instance the shortest one) of each vector, and for encoding negative numbers as their sign followed by the encoding or their absolute value. It turns out that these alternate choices complicate substantially some elementary manipulation algorithms, such as computing the Cartesian product or the difference of two sets, as well as the construction of the automata representing atomic formulas, such as linear equations or inequations. On the other hand, our present choices lead to simple manipulation algorithms, with the only exceptions of projection and counting.

3 Projecting NDDs

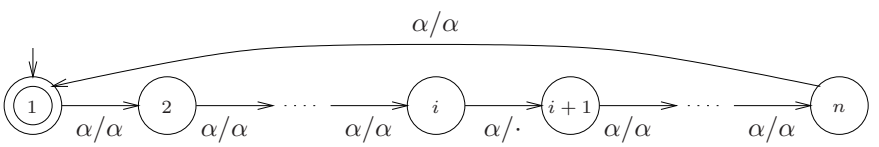
The projection problem can be stated in the following way. Given an NDD $\mathcal{A}$ representing a set $S \subseteq \mathbf{Z}^n$, with $n > 0$, and a component number $i \in \{1, \ldots, n\}$, the goal is to construct an NDD $\mathcal{A}'$ representing the set

$$\exists_i S = \{(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \mid (v_1, \ldots, v_n) \in S\}.$$

For every accepting path of $\mathcal{A}$, there must exist a matching path of $\mathcal{A}'$, from the label of which the digits corresponding to the i-th vector component are excluded. Thus, one could be tempted to compute $\mathcal{A}'$ as the direct result of applying to $\mathcal{A}$ the transducer depicted at Figure 1.

Unfortunately, this method produces an automaton $\mathcal{A}|_{\neq i}$ that, even though it accepts valid encodings of all the elements of $\exists_i S$, is generally not an NDD. Indeed, for some vectors, the automaton may only recognize their encodings if they are of sufficient length, think for instance of $\exists_1\{(4, 1)\}$. In order to build $\mathcal{A}'$ from $\mathcal{A}|_{\neq i}$, one thus has to transform the automaton so as to make it also accept the shorter encodings of the vectors that it recognizes.

Clearly, two encodings of the same vector only differ in the number of times that their sign header is repeated. We can thus restate the previous problem in the following way: Given a finite-state automaton $\mathcal{A}_1$ of alphabet Σ accepting the language L_1, and a dimension $n \geq 0$, construct an automaton $\mathcal{A}_2$ accepting $L_2 = \{u^i w \mid u \in \{0, r-1\}^n \wedge w \in \Sigma^* \wedge i \in \mathbf{N} \wedge (\exists k > 0)(k \geq i \wedge u^k w \in L_1)\}$.



For all transitions, $\alpha \in \{0, \ldots, r-1\}$.

Fig. 1. Projection transducer

In [Boi99], this problem is solved by considering explicitly every potential value u of the sign header, and then exploring $\mathcal{A}_1$ in order to know what states can be reached by a prefix of the form u^i, with $i > 0$. It is then sufficient to make each of these states reachable after reading a single occurrence of u, which can be done by a simple construction, and to repeat the process for other u. Although satisfactory from a theoretical point of view, this solution exhibits a systematic cost in $O(2^n)$ which limits its practical use to problems with a very small vector dimension.

The main idea behind our improved solution consists of handling simultaneously sign headers that cannot be distinguished from each other by the automaton $\mathcal{A}_1$, i.e., sign headers $u_1, u_2 \in \{0, r-1\}^n$ such that for every $k > 0$, reading u_1^k leads to the same automaton states as reading u_2^k. For simplicity, we assume $\mathcal{A}_1$ to be deterministic[1].

Our algorithm proceeds as follows. First, it extracts from $\mathcal{A}_1$ a *prefix automaton* $\mathcal{A}_P$ that reads only the first n symbols of words and associates one distinct end state to each group of undistinguished sign headers. Each end state of $\mathcal{A}_P$ is then matched to all the states of $\mathcal{A}_1$ that can be reached by reading the corresponding sign headers any number of times. At every time during this operation one detects two sign headers that are not yet distinguished but that lead to different automaton states, one refines the prefix automaton $\mathcal{A}_P$ so as to associate a different end state to each header. Finally, the automaton $\mathcal{A}_2$ is constructed in such a way that following one of its accepting paths amounts to reading n symbols in $\mathcal{A}_P$, which results in reaching an end state s of this automaton, and then following an accepting path of $\mathcal{A}_1$ starting from a state matched to s.

The algorithm is formally described in Appendix A. Its worst-case time complexity is not less than that of the simple solution [Boi99] outlined at the beginning of this section. However, in the context of state-space exploration applications, we observed that it succeeds most of the time, if not always, to avoid the exponential blowup experienced with the latter approach.

4 Counting Elements of NDDs

We now address the problem of counting the number of vectors that belong to a set S represented by an NDD $\mathcal{A}$. Our solution proceeds in two steps: First, we check whether S is finite or infinite and, in the former case, we transform $\mathcal{A}$ into a deterministic automaton $\mathcal{A}'$ that accepts exactly one encoding of each vector that belongs to S. Second, we count the number of distinct accepting paths in $\mathcal{A}'$.

4.1 Transformation Step

Let $\mathcal{A}$ be an NDD representing the set $S \subseteq \mathbf{Z}^n$. If S is not empty, then the language accepted by $\mathcal{A}$ is infinite, hence the transition graph of this automaton

[1] This is not problematic in practice, since the cost of determinizing an automaton built from an arithmetic formula is often moderate [WB00].

contains cycles. In order to check whether S is finite or not, we thus have to determine if these cycles are followed when reading different encodings of the same vectors, or if they can be iterated in order to recognize an infinite number of distinct vectors.

Assume that $\mathcal{A}$ does not contain unnecessary states, i.e., that all its states are reachable and that there is at least one accepting path starting from each state. We can classify the cycles of $\mathcal{A}$ in three categories:

- A *sign loop* is a cycle that can only be followed while reading the sign header of an encoding, or a repetition of that sign header;
- An *inflating loop* is a cycle that can never be followed while reading the sign header of an encoding or one of its repetitions;
- A *mixed loop* is a cycle that is neither a sign nor an inflating loop.

If $\mathcal{A}$ has at least one inflating or mixed loop, then one can find an accepting path in which one follows that loop while not reading a repetition of a sign header. By iterating the loop, one thus gets an infinite number of distinct vectors, which results in S being infinite. The problem thus reduces to checking whether $\mathcal{A}$ has non-sign (i.e., inflating or mixed) loops[2]. Thanks to the following result, this check can be carried out by inspecting the transition graph of $\mathcal{A}$ without paying attention to the transition labels.

Theorem 1. *Assume that $\mathcal{A}$ is a deterministic and minimal (with respect to language equivalence) NDD. A cycle λ of $\mathcal{A}$ is a sign loop if and only if it can only be reached by one path (not containing any occurrence of that cycle).*

Proof. Since $\mathcal{A}$ is an NDD, it can only accept words whose length is a multiple of n. The length of λ is thus a multiple of n.

- *If λ is reachable by only one path π.* Let $u \in \{0, r-1\}^n$ be the sign header that is read while following the n first transitions of the path $\pi\lambda$, and let s and s' be the states of $\mathcal{A}$ respectively reached after reading the words u and uu (starting from the initial state).
  Since $\mathcal{A}$ accepts all the encodings of the vectors in S, it accepts, for every $w \in \{0, 1, \ldots, r-1\}^*$, the word uw if and only if it accepts the word uuw. It follows that the languages accepted from the states s and s' are identical which implies, since $\mathcal{A}$ is minimal, that $s = s'$.
  Therefore, λ can only be visited while reading the sign header u or its repetition, and is thus a sign loop.
- *If λ is reachable by at least two paths π_1 and π_2.* Let kn, with $k \in \mathbf{N}$ be the length of λ. Since $\mathcal{A}$ only accepts words whose length is a multiple of n, there are exactly k states $s_1, s_2, \ldots, s_k$ that are reachable in λ from the initial state of $\mathcal{A}$ after following a multiple of n transitions.
  If the words read by following λ from s_1 to s_2, from s_2 to s_3, ..., and from s_k to s_1 are not all identical, then λ is not a sign loop. Otherwise, let u^k, with $u \in \{0, 1, \ldots, r-1\}^n$, be the label of λ.

[2] An example of a non-trivial instance of this problem can be obtained by building the minimal deterministic NDD representing the set $\{(x, y) \in \mathbf{Z}^2 \mid x + y \leq 0 \wedge x \geq 0\}$.

Since $\mathcal{A}$ is deterministic, at least one of the blocks of n consecutive digits read while following π_1 or π_2 up to reaching λ differs from u. Thus, λ can be visited while not reading a repetition of a sign header. □

Provided that $\mathcal{A}$ has only sign loops, it can easily be transformed into an automaton $\mathcal{A}'$ that accepts exactly one encoding of each vector in S by performing a depth-first search in which one removes, for each detected cycle, the transition that gets back to a state that has already been visited in the current exploration path. This operation does not influence the set of vectors recognized by the automaton, since the deleted transitions can only be followed while reading a repeated occurrence of a sign header.

An algorithm that combines the classification of cycles with the transformation of $\mathcal{A}$ into $\mathcal{A}'$ is given in Appendix B. Since each state of $\mathcal{A}$ has to be visited at most once, the time and space costs of this algorithm – if suitably implemented – are linear in the number of states of $\mathcal{A}$.

4.2 Counting Step

If S is finite, then the transition graph of the automaton $\mathcal{A}'$ produced by the algorithm given in the previous section is acyclic. The number of vectors in S corresponds to the number of accepting paths originating in the initial state of $\mathcal{A}'$.

For each state s of $\mathcal{A}'$, let $N(s)$ denote the number of paths of $\mathcal{A}'$ that start at s and end in an accepting state. Each of these paths either leaves s by one of its outgoing transitions, or has a zero length (which requires s to be accepting). Thus, we have at each state s

$$N(s) = \sum_{(s,d,s') \in \Delta} N(s') + acc(s),$$

where $acc(s)$ is equal to 1 if s is accepting, and to 0 otherwise.

Thanks to this rule, the value of $N(s)$ can easily be propagated from the states that have no successors to the initial state of $\mathcal{A}'$, following the transitions backwards. The number of additions that have to be performed is linear in the number of states of $\mathcal{A}'$.

5 Example of Use

The projection and counting algorithms presented in Sections 3 and 4 have been implemented in the verification tool LASH [LASH], whose main purpose is to compute exactly the set of reachable configurations of a system with finite control and unbounded data. Sketchily, this tool handles finite and infinite sets of configurations with the help of finite-state representations suited for the corresponding data domains, and relies on *meta-transitions*, which capture the repeated effect of control loops, for exploring infinite state spaces in finite time. A description of the main techniques implemented by LASH is given in [Boi99].

In the context of this paper, we focus on systems based on unbounded integer variables, for which the set representation system used by LASH is the NDD. Our present results thus allow to count precisely the number of reachable system configurations that belong to a set computed by LASH.

Let us now describe an example of a state-space exploration experiment featuring the counting algorithm. We consider the simple lift controller originally presented in [Val89]. This system is composed of two processes modeling a lift panel and its motor actuator, communicating with each other by means of shared integer variables. A parameter N, whose value is either fixed in the model or left undetermined, defines the number of floors of the building. In the former case, one observes that the amount of time and of memory needed by LASH in order to compute the set of reachable configurations grows only logarithmically in N, despite the fact that the number of elements in this set is obviously at least $O(N^2)$. (Indeed, the behavior of the lift is controlled by two main variables modeling the current and the target floors, which are able to take any pair of values in $\{1, \ldots, N\}^2$.)

Our simple experiment has two goals: Studying precisely the evolution of the number of reachable configurations with respect to increasing values of N, and evaluating the amount of acceleration induced by meta-transitions in the state-space exploration process.

The results are summarized in Figures 2 and 3. The former table gives, for several values of N, the size (in terms of automaton states) of the finite-state representation of the reachable configurations, the exact number of these configurations, and the total time needed to perform the exploration. These results clearly show an evolution in $O(N^2)$, as suspected. It is worth mentioning that, thanks to the fact that the cost of our counting algorithm is linear in the size of NDDs, its execution time (including the classification of loops) was negligible with respect to that of the exploration.

The latter table shows, for $N = 10^9$, the evolution of the number of configurations reached after the successive steps of the exploration algorithm. Roughly speaking, the states are explored in a breadth-first fashion, starting from the initial configuration and following transitions as well as meta-transitions, until a fixpoint is detected. In the present case, the impact of meta-transitions on the number of reached states is clearly visible at Steps 2 and 4.

N	NDD states	Configurations	Time (s)
10	852	930	25
100	1782	99300	65
1000	2684	9993000	101
10000	3832	999930000	153
100000	4770	99999300000	196
1000000	5666	9999993000000	242

Fig. 2. Number of reachable configurations w.r.t. N

Step	NDD states	Configurations
1	638	3
2	1044	1000000003
3	1461	3999999999
4	2709	500000005499999997
5	4596	1500000006499999995
6	6409	3500000004499999994
7	7020	6499999997499999999
8	7808	7999999995000000000
9	8655	8999999994000000000
10	8658	9499999993500000000
11	8663	9999999993000000000

Fig. 3. Number of reached configurations w.r.t. exploration steps

6 Conclusions and Comparison with Other Work

The main contribution of this paper is to provide an algorithm for counting the number of elements in a set represented by an NDD. As an auxiliary result, we also present an improved projection algorithm that makes it possible to build efficiently an NDD representing the set of solutions of a Presburger formula. Our algorithms have been implemented in the tool LASH.

The problem of counting the number of solutions of a Presburger equation has already been addressed in [Pug94], which follows a formula-based approach. More precisely, that solution proceeds by decomposing the original formula into an union of disjoint convex sums, each of them being a conjunction of linear inequalities. Then, all but one variable are projected out successively, by splitering the sums in such a way that the eliminated variables have one single and one upper bound. This eventually yields a finite union of simple formulas, on which the counting can be carried out by simple rules.

The main difference between this solution and ours is that, compared to the general problem of determining whether a Presburger formula is satisfiable, counting with a formula-based method incurs a significative additional cost. On the other hand, the automata-based counting method has no practical impact on the execution time once an NDD has been constructed. Our method is thus efficient for all the cases in which an NDD can be obtained quickly, which, as it has been observed in [BC96, WB00], happens mainly when the coefficients of the variables are small. In addition, since automata can be determinized and minimized after each manipulation, NDDs are especially suited for representing the results of complex sequences of operations producing simple sets, as in most state-space exploration applications. The main restriction of our approach is that it cannot be generalized in a simple way to the more complex counting problems, such as summing polynomials over Presburger-definable sets, that are addressed in [Pug94].

References

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of CAAP'96*, number 1059 in Lecture Notes in Computer Science, pages 30–43. Springer-Verlag, 1996. 41, 48

[BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society*, 1(2):191–238, March 1994. 42

[Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. Collection des publications de la Faculté des Sciences Appliquées de l'Université de Liège, Liège, Belgium, 1999. 41, 42, 44, 46

[Cob69] A. Cobham. On the base-dependence of sets of numbers recognizable by finite automata. *Mathematical Systems Theory*, 3:186–192, 1969. 42

[FO97] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Proceedings of CONCUR'97*, volume 1243, pages 213–227, Warsaw, Poland, July 1997. Springer-Verlag. 40

[KSW95] F. Kabanza, J.-M. Stevenne, and P. Wolper. Handling infinite temporal data. *Journal of computer and System Sciences*, 51(1):3–17, 1995. 40

[LASH] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`. 41, 46

[PR96] M. Padberg and M. Rijal. *Location, Scheduling, Design and Integer Programming*. Kluwer Academic Publishers, Massachusetts, 1996. 40

[Pre29] M. Presburger. Über die Volständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 92–101, Warsaw, Poland, 1929. 40

[Pug92] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, pages 102–114, August 1992. 40

[Pug94] W. Pugh. Counting solutions to Presburger formulas: How and why. *SIGPLAN*, 94-6/94:121–134, 1994. 41, 48

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & sons, Chichester, 1986. 40

[Sem77] A. L. Semenov. Presburgerness of predicates regular in two number systems. *Siberian Mathematical Journal*, 18:289–299, 1977. 42

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Proceedings of the 10th Intl. Conf. on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, Vancouver, June/July 1998. Springer-Verlag. 40

[Val89] A. Valmari. State space generation with induction. In *Proceedings of the SCAI'89*, pages 99–115, Tampere, Finland, June 1989. 47

[WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32, Glasgow, September 1995. Springer-Verlag. 41

[WB00] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag. 44, 48

A Projection Algorithm

Let $(\Sigma, Q, s^{(0)}, \Delta, F)$ be the automaton $\mathcal{A}_1$, where Σ is the alphabet $\{0, \ldots, r-1\}$, Q is a finite set of states, $s^{(0)} \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is a set of accepting states.

1. Let $\mathcal{A}_P = (\Sigma, Q_P, s_P^{(0)}, \Delta_P, F_P)$, with $s_P^{(0)} = (s^{(0)}, 0)$, $Q_P = \{s_P^{(0)}\}$, and $\Delta_P = F_P = \emptyset$. Each state (s, i) of $\mathcal{A}_P$ is composed of a state s of $\mathcal{A}_1$ associated with an index i ranging from 0 to n. The index n corresponds to the end states.
2. For $i = 1, \ldots, n$ and for each $(s, \alpha, s') \in \Delta$ such that $(s, i-1) \in Q_P$, add (s', i) to Q_P and $((s, i-1), \alpha, (s', i))$ to Δ_P.
3. For each $s \in Q$ such that $(s, n) \in Q_P$, let $\mathit{matches}[(s, n)] = \{s\}$.
4. Let $\mathit{remaining} = \{(s, s) \mid (s, n) \in Q_P\}$.
5. For each $(s, s') \in \mathit{remaining}$:
   - If there does not exist $s'' \in Q \setminus \mathit{matches}[(s, n)]$ and $u \in \Sigma^n$ such that $(s_P^{(0)}, u, (s, n)) \in \Delta_P^*$ and $(s', u, s'') \in \Delta^*$, then remove (s, s') from *remaining*.
   - If there exists $s'' \in Q \setminus \mathit{matches}[(s, n)]$ such that for every $u \in \Sigma^n$ for which $(s_P^{(0)}, u, (s, n)) \in \Delta_P^*$, $(s', u, s'') \in \Delta^*$, then add s'' to the set $\mathit{matches}[(s, n)]$, add (s, s'') to *remaining*, and remove (s, s') from *remaining*.
   - Otherwise, find $u, u' \in \Sigma^n$ such that $(s_P^{(0)}, u, (s, n)) \in \Delta_P^*$, $(s_P^{(0)}, u', (s, n)) \in \Delta_P^*$ and either
     - there exists $s'', s''' \in Q$, $s'' \neq s'''$, such that $(s', u, s'') \in \Delta^*$ and $(s', u', s''') \in \Delta^*$, or
     - there exists $s'' \in Q$ such that $(s', u, s'') \in \Delta^*$ but no $s''' \in Q$ such that $(s', u', s''') \in \Delta^*$,

     then refine $\mathcal{A}_P$ with respect to the state s' and the headers u and u' (this operation will be described separately).
6. Let $\mathcal{A}_2 = (\Sigma, Q_2, s_2^{(0)}, \Delta_2, F_2)$, with $Q_2 = Q \cup Q_P$, $s_2^{(0)} = s_P^{(0)}$, $\Delta_2 = \Delta \cup \Delta_P \cup \{((s, n), \varepsilon, s') \mid s' \in \mathit{matches}[(s, n)]\}$, and $F_2 = F$.

It is worth mentioning that the test performed at Line 5 can be carried out efficiently by a search in the transition graph of the automata. The details of this operation are omitted from this short description.

A central step of the algorithm consists of refining the prefix automaton $\mathcal{A}_P$ in order to associate different end states to two sign headers u and u' read from the state s' of $\mathcal{A}_1$. This operation is performed as follows:

1. Let $k \in \{1, \ldots, n\}$ be the smallest integer such that the paths reading u and u' from the state $s_P^{(0)}$ of $\mathcal{A}_P$ reach the same state after having followed k transitions, and the paths reading u and u' from the state s' of $\mathcal{A}_1$ reach two distinct states after the same number k of transitions.
2. Let $((s_1, k-1), d, (s_2, k))$ and $((s_1', k-1), d', (s_2, k))$ be the k-th transitions of the paths reading (respectively) u and u' in $\mathcal{A}_P$.

3. For each $q \in Q_P$ such that $((s_2, k), w, q) \in \Delta_P^*$ for some $w \in \Sigma^*$, add a new state q' to Q_P and set $split[q] = q'$.
4. For each transition $(q, d, q') \in \Delta_P$ such that $split[q]$ is defined, add the transition $(split[q], d, split[q'])$ to Δ_P.
5. Replace the transition $((s_1', k-1), d', (s_2, k))$ by $((s_1', k-1), d', split[(s_2, k)])$ in Δ_P.
6. For each $q \in Q_P$ such that $split[q]$ exists, let $matches[split[q]] = matches[q]$.
7. For each $(s, s') \in remaining$ such that $split[(s, n)]$ is defined, add the pair $(split[(s, n)], s')$ to $remaining$.

B Cycle Classification and Removal Algorithm

1. Let $\mathcal{A} = (\Sigma, Q, s^{(0)}, \Delta, F)$, let $visited = \emptyset$, and for each state $s \in Q$, let $leads\text{-}to\text{-}cycle[s] = \mathbf{F}$;
2. If $explore(s^{(0)}, 0) = \mathbf{F}$, then the set represented by $\mathcal{A}$ is infinite. Otherwise, the automaton $\mathcal{A}'$ is given by $(\Sigma, Q, s^{(0)}, \Delta, F)$.

Subroutine $explore(s, k)$:

1. Let $visited = visited \cup \{s\}$, and let $history[k] = s$;
2. For each $(s', d, s'') \in \Delta$ such that $s' = s$:
   - If $s'' \notin visited$, then
     (a) If $explore(s'', k+1) = \mathbf{F}$ then return $\mathbf{F}$;
     (b) If $leads\text{-}to\text{-}cycle[s'']$ then let $leads\text{-}to\text{-}cycle[s] = \mathbf{T}$;
   - If $(\exists i < k)(history[i] = s'')$, then
     (a) If $leads\text{-}to\text{-}cycle[s]$ then return $\mathbf{F}$;
     (b) Let $leads\text{-}to\text{-}cycle[s] = \mathbf{T}$, and remove (s', d, s'') from Δ;
   - If $s'' \in visited$ and $(\forall i < k)(history[i] \neq s'')$, then
     (a) If $leads\text{-}to\text{-}cycle[s'']$ then return $\mathbf{F}$;
3. Return $\mathbf{T}$.

Brzozowski's Derivatives Extended to Multiplicities*

Jean-Marc Champarnaud and Gérard Duchamp

University of Rouen, LIFAR
{champarnaud,duchamp}@univ-rouen.fr

Abstract. Our aim is to study the set of K-rational expressions describing rational series. More precisely we are concerned with the definition of quotients of this set by coarser and coarser congruences which lead to an extension – in the case of multiplicities – of some classical results stated in the Boolean case. In particular, analogues of the well known theorems of Brzozowski and Antimirov are provided in this frame.

1 Introduction

Language theory is a rich and everlasting domain of study since computers have always been operated by identifiers and sequences of words. In the case when weights are associated to words, the theory of series, which is an extension of language theory, is invoked. Some results of the two theories are strikingly similar, the proeminent example being the theorem of Kleene-Schützenberger which states that a series is rational if and only if it is recognizable (by a K-automaton) [25]. Therefore, we feel that it should be of interest to contribute to build firm foundations to the study of abstract formulae (i.e. K-rational expressions) describing rational series. These formulae have been used as a powerful tool to describe the inverse of a noncommutative matrix [12].
Rational expressions are realizable into the algebra of series. They are the counterpart of regular expressions of language theory and our work on rational expressions is close to the contributions of Antimirov [1], Brzozowski [4] and more recently Champarnaud and Ziadi [7, 8, 9] who study the properties of regular expressions and their derivatives.
The kernel of the projection: *rational expressions* $\rightarrow$ *rational series* will be called $\sim_{rat}$. We are concerned here with the study of congruences which are finer than $\sim_{rat}$ and which give rise to normal forms (for references on the subject of rational identities see [3, 5, 17, 23]). Antimirov in [1] gives a list of axioms suited to the Boolean case. We give here a list of K-axioms which will be treated as congruences, extending the preceding ones in the case of multiplicities. A set of coarser and coarser congruences is considered and analogues of the well known theorems of Antimirov [1] and Brzozowski [4] are provided in this frame.

The structure of the paper is the following. The main theorems concerning congruences on the set of regular expressions are gathered in the next section.

* Partially supported by the MENRT Scientific Research Program ACI Cryptology.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 52–64, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Section 3 gives a brief description of formal series and rational expressions. Section 4 introduces the notion of K-module congruence, provides a list of admissible congruences to compute with rational expressions and states an analogue of Antimirov's theorem in the setting of multiplicities. Section 5 deals with the existence of deterministic recognizers and gives a generalization of Brzozowski's theorem.

2 Regular Expressions

We briefly recall results issued from the works of Brzozowski [4] and Antimirov [1] in the Boolean domain. The reader is referred to [27] for a recent survey of automaton theory.

Brzozowski has defined the notion of *word derivative* of a regular expression. Let $\mathcal{R}(\Sigma)$ be the set of regular expressions over a given alphabet Σ. Let 0 denote the null expression and ε the empty word. Let E, F and G be regular expressions. We consider the following congruences on $\mathcal{R}(\Sigma)$:

- $E + (F + G) \sim (E + F) + G$ (Associativity of $+$)
- $E + F \sim F + E$ (Commutativity of $+$)
- $E + E \sim E$ (Idempotency of $+$)

The $\sim_{aci}$ congruence is defined by [A,C,I].

Theorem 1 (Brzozowski). *The set of derivatives of every regular expression in $\mathcal{R}(\Sigma)/\sim_{aci}$ is finite.*

Antimirov has introduced the notion of *partial derivative* of a regular expression. A *monomial* is a pair $< x, E >$ where x is a symbol of Σ and E a non null regular expression. A linear form is a set of monomials. The word concatenation is extended to linear forms by the following equations, where l and l' are arbitrary linear forms, and F and E are regular expressions different of 0 and of ε:

$$
\begin{aligned}
l \odot 0 &= \emptyset \\
\emptyset \odot E &= \emptyset \\
l \odot \varepsilon &= l \\
\{< x, \varepsilon >\} \odot E &= \{< x, E >\} \\
\{< x, F >\} \odot E &= \{< x, F \cdot E >\} \\
(l \cup l') \odot E &= (l \odot E) \cup (l' \odot E)
\end{aligned}
$$

The linear form $lf(E)$ of a regular expression E is the set of monomials inductively defined as follows:

$$
\begin{aligned}
lf(0) &= \emptyset \\
lf(\varepsilon) &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
lf(x) &= \{< x, \varepsilon >\}, \forall x \in \Sigma \\
lf(F+G) &= lf(F) \cup lf(G) \\
lf(F \cdot G) &= \begin{cases} lf(F) \odot G & \text{if } Null(F) = 0 \\ lf(F) \odot G \cup lf(G) & \text{otherwise} \end{cases} \\
lf(F^*) &= lf(F) \odot F^*
\end{aligned}
$$

Given a linear form $l = \{< x_1, F_1 >, ..., < x_k, F_k >\}$ we write $\sum fl(E)$ to denote the regular expression $x_1 \cdot F_1 + ... + x_k \cdot F_k$ (up to an arbitrary permutation of the summands). Notice that $\sum \emptyset$ is 0.

Theorem 2 (Antimirov). *For any regular expression E in $\mathcal{R}(\Sigma)$, the following linear factorization holds:*

$$
E = \begin{cases} \sum lf(E) & \text{if } Null(E) = 0 \\ \varepsilon + \sum lf(E) & \text{otherwise} \end{cases}
$$

Finally, F is a partial derivative of E w.r.t. x if and only if there exists a monomial $< x, F >$ in $lf(E)$. The following result holds:

Theorem 3 (Antimirov). *The set of partial derivatives of every regular expression in $\mathcal{R}(\Sigma)$ is finite.*

3 Series and Rational Expressions

3.1 Noncommutative Formal Series (NFS)

The Algebra of NFS We give here a brief description of the, by now classical, theory of series. The reader is also invited to consult [3, 14, 26].

A semiring $K(+, \times)$ is the data of two structures of monoid $(K, +)$ (commutative) and $(K, \times)$ (not necessarily commutative), $\times$ being distributive over $+$ and 0_K being an annihilator (roughly speaking, a semiring is a ring where the "minus" operation may not exist). For a set of symbols Σ, a NFS is a mapping $f : \Sigma^* \to K$. The set of NFS (i.e. K^{Σ^*}) is often denoted $K\langle\langle\Sigma\rangle\rangle$. One denotes alternatively f in the "sum-like" form $S = \sum_{w \in \Sigma^*} f(w)w$ which appeals, in a natural way, the scalar product denotation $f(w) = \langle S|w\rangle$.

For every family of series $(S_i)_{i \in I}$, if for each word $w \in \Sigma^*$ the mapping $i \to \langle S_i|w\rangle$ has a finite support (i.e. the set of indices for which $\langle S_i|w\rangle \neq 0$ is finite), then the series:

$$
\sum_{w \in \Sigma^*} \left(\sum_{i \in I} \langle S_i|w\rangle \right) w
$$

is well-defined and will be denoted by $\sum_{i \in I} S_i$. Such a family $(S_i)_{i \in I}$ will be called summable.

The following operations are natural in $K\langle\langle\Sigma\rangle\rangle$. Let us recall them:

1. Sum and scalings are defined componentwise:
$$\sum_{w\in\Sigma^*} f(w)w + \sum_{w\in\Sigma^*} g(w)w := \sum_{w\in\Sigma^*} (f(w)+g(w))w$$
$$\lambda \sum_{w\in\Sigma^*} f(w)w := \sum_{w\in\Sigma^*} (\lambda f(w))w;\ \left(\sum_{w\in\Sigma^*} f(w)w\right)\lambda := \sum_{w\in\Sigma^*} (f(w)\lambda)w$$
2. Concatenation, Cauchy product, or convolution:
$$\left(\sum_{w\in\Sigma^*} f(w)w\right).\left(\sum_{w\in\Sigma^*} g(w)w\right) := \sum_{w\in\Sigma^*}\left(\sum_{uv=w} f(u)g(v)\right)w$$
3. If S is without constant term (i.e. $\langle S|\varepsilon\rangle = 0_K$), the family $(S^n)_{n\in\mathbb{N}}$ is summable, and the sum $\sum_{n\geq 0} S^n$ will be denoted S^*.

Now, we get an algebra with four binary laws, two external ones (scalings) and two internal ones (sum and concatenation) and a unary internal law partially defined (the star). Notice that, when K is commutative, with f, λ as above, one has $\lambda.f = f.\lambda$ and only the left action of K is required.

The adjoint operation of left and right multiplications can be called shifts (known sometimes as "quotients" see [14]) and is of the first importance for the study of rationality. One can use a covariant denotation (such as $u \triangleleft f;\ f \triangleright u$) or a contravariant one (such as $u^{-1}f;\ fu^{-1}$).

Definition 1. *A) Right shifts (left quotients) of* $S := \sum_{w\in\Sigma^*}\langle S|w\rangle w$ *are defined by*
$$\langle S \triangleright u|w\rangle = \langle S|uw\rangle = \langle u^{-1}S|w\rangle$$
B) Left shifts (right quotients) of $S := \sum_{w\in M}\langle S|w\rangle w$ *are defined by*
$$\langle u \triangleleft S|w\rangle = \langle S|wu\rangle = \langle Su^{-1}|w\rangle$$

Note 1. i) It is easy to see that "triangle" is covariant: $(S\triangleright u)\triangleright v = S\triangleright uv;\ u\triangleleft(v\triangleleft S) = uv\triangleleft S$, and "quotient" is contravariant: $u^{-1}(v^{-1}S) = vu^{-1}S;\ (Su^{-1})v^{-1} = S(vu)^{-1}$.
ii) Shifts are (two-sided) linear, they satisfy very simple identities. Let $a \in \Sigma;\ S,\ S_i \in K\langle\langle\Sigma\rangle\rangle\ (i = 1, 2)$. The following identities hold:
$$\begin{aligned} a^{-1}x &= \varepsilon \text{ if } x = a \\ &= 0 \text{ if } x \in (\Sigma - \{a\}) \cup \{0\} \\ a^{-1}(S_1 + S_2) &= a^{-1}S_1 + a^{-1}S_2 \\ a^{-1}(\lambda S) &= \lambda a^{-1}S;\ a^{-1}(S\lambda) = (a^{-1}S)\lambda; \\ a^{-1}(S_1.S_2) &= (a^{-1}S_1).S_2 + const(S_1)a^{-1}(S_2) \\ a^{-1}(S^*) &= (a^{-1}S).S^* \text{ (if } S \text{ has a null constant term)} \end{aligned}$$
Notice that similar identities hold for the trace monoid [11].
iii) Right shifts commute with left shifts (straightforwardly due to associativity) and satisfy similar identities.

Example 1. For example, with $a \in \Sigma;\ \alpha, \beta \in K$ and $S = (a\alpha)^*(\beta a)^*$ one has $(a^{-1})^2 S = a^{-2}E = \alpha^2 S + (\alpha\beta + \beta^2)(\beta a)^*$. Finally, we get:

3.2 Rational Expressions

Construction, Constant Terms and Shifts The completely free formulas for these laws is the universal algebra generated by $\Sigma \cup \{0_{\mathcal{E}}\}$ as constants and the five preceding laws ($1_{\mathcal{E}}$ will be constructed as $0_{\mathcal{E}}^*$ and still be denoted ε). These expressions, by a standard argument form a set which will be denoted $\mathcal{E}^{cf}(\Sigma, K)$.

Example 2. For example $(a^*)^* \in \mathcal{E}^{cf}(\Sigma, K)$. However, we will see later that this expression is not to be considered as valid in our setting.

Now, we construct a pull-back of the "constant term" mapping of the series.

Definition 2. *i) The function const* : $\mathcal{E}^{cf}(\Sigma, K) \to K$ *is (partially) recursively defined by the rules:*

1. *If* $x \in \Sigma \cup \{0_{\mathcal{E}}\}$ *then* $const(x) = 0_K$.
2. *If* $E, E_i \in \mathcal{E}^{cf}(\Sigma, K),\ i = 1, 2$ *then*

$$const(E_1 + E_2) = const(E_1) + const(E_2),\ const(E_1 \cdot E_2) = const(E_1) \times const(E_2)$$
$$const(\lambda E) = \lambda const(E), const(E\lambda) = const(E)\lambda.$$

3. *If* $const(E) = 0_K$ *then* $const(E^*) = 1_K$.

ii) The domain of const (i.e. the set of expressions for which const is defined) will be denoted $\mathcal{E}(\Sigma, K)$ *(or* $\mathcal{E}$*, for short), in the sequel (we then have* $(0_K)^* = \varepsilon \in \mathcal{E}$*).*

Remark 1. i) We define left and right shifts by formulas of the Note 1 and their right analogues.
In this way, it is easy to see that we get well (everywhere) defined operators on $\mathcal{E}(\Sigma, K)$ which will be still denoted $a^{-1}(?)$ and $(?)a^{-1}$ in the sequel.
ii) The set $\mathcal{E}(\Sigma, \mathbb{B})$ is a strict subset of the set of free regular expressions, but due to the (Boolean) identity $(X + \varepsilon)^* = X^*$, the two sets have the same expressive power.
iii) The class of rational expressions is a small set (in the sense of Mc Lane [20]), its cardinal is countable if Σ and K are finite or countable.
iv) Sticking to our philosophy of "following the Boolean track", we must be able to evaluate rational expressions within the algebra of series. It is a straightforward verification to see that, given a mapping $\phi : \Sigma \to \Sigma^+$, there exists a unique (poly)morphism $\bar{\phi} : \mathcal{E} \to K\langle\langle\Sigma\rangle\rangle$ which extends ϕ. In particular, let $\phi : \Sigma \to \Sigma^+$ be the inclusion mapping, then the kernel of $\bar{\phi}$ will be denoted $\sim_{rat}$. Notice here that $\bar{\phi}(1_{\mathcal{E}}) = \varepsilon$.

Now, we can state a celebrated theorem which is coined as Kleene-Schützenberger's theorem.

Theorem 4. *For a series* $S \in K\langle\langle\Sigma\rangle\rangle$*, the following conditions are equivalent:*
i) The series S *is in the image of* $\bar{\phi}$.
ii) There exists a finite family $(S_i)_{i \in I}$*, stable by derivation (i.e.* $(\forall i \in I)(\forall a \in \Sigma)\ a^{-1}S_i = \sum_{j \in I} \mu_{ij}(a)S_j$*) such that* S *is a linear combination of the* S_i *(i.e.* $S = \sum_{i \in I} \lambda_i S_i$*).*

Definition 3. *A series which fulfills the preceding equivalent conditions will be called rational. The set of rational series is denoted* $K^{\mathrm{rat}}\langle\langle\Sigma\rangle\rangle$.

Congruences We are now interested to describe series by quotient structures of $\mathcal{E}(\Sigma, K)$ (going from $\mathcal{E}(\Sigma, K) \cong \mathcal{E}(\Sigma, K)/= \text{ to } K^{\mathrm{rat}}\langle\langle\Sigma\rangle\rangle \cong \mathcal{E}(\Sigma, K)/\sim_{rat}$). If the equivalence is $\sim_{rat}$, we get the series with the advantage of algebraic facilities (K-module structures, many identities, etc...) but syntactic difficulties. In fact, the equivalence $\sim_{rat}$ is not well understood (the question of systems of identities - on expressions - for the K-algebra of series has been discussed in [5, 17]). On the other end, the equality does not provide even the identity: $\lambda(E+F) \sim \lambda E + \lambda F$ or, at least, associativity. This is the reason why Brzozowski [4] and Antimirov [1] studied intermediate congruences. What follows is a step in this direction.

Definition 4. *A congruence on the algebra* $\mathcal{E}(K, \Sigma)$ *is an equivalence* $\sim$ *which is compatible with the laws (i.e. with subtrees substitution).*

The following proposition is rather straightforward, but of crucial importance.

Proposition 1. *The set of congruences on* $\mathcal{E}(\Sigma, K)$ *is a complete sublattice of the lattice of all equivalence relations.*

At this level, three things are lacking. First, rational expressions do not yet form a K-module in spite of the fact that the operators a^{-1} are wanted to be linear; second, an expression can have infinitely many independent derivatives (for example $E = (a^*).(a^*)$ with $K = \mathbb{N}$) and to end with we do not recover Brzozowski's theorem. There is a simple way to solve this at once. It consists in associating the expressions which are identical "up to a K-module axiom"; these congruences will be called K-module congruences.

4 *K*-module Congruences

From now on, for a lighter exposition, we will consider K as a commutative semiring. For K noncommutative the theory holds but needs the structure of $K-K$-bimodule with is rather cumbersome to expound (and therefore confusing at first sight).
We shall see that there is a finest congruence $\sim_{acm1}$ such that the quotients of the laws $+ : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ and $._{ext} : K \times \mathcal{E} \to \mathcal{E}$ endow $\mathcal{E}/\sim$ with a K-module structure. But, to get the classical "step by step" construction which guarantees that every rational expression can be embedded into a finite type module, one needs a little more (i.e. $\sim_{acm2}$).

4.1 General Definitions and Properties

Definition 5. *Let* $(M, +)$ *be a commutative monoid with neutral* 0_M. *A* K*-module structure on* M*, is the datum of an external law* $K \times M \to M$ *satisfying identically:*

1. $\lambda(u+v) = \lambda u + \lambda v;\ \lambda 0_M = 0_M$
2. $(\lambda+\mu)u = \lambda u + \mu u;\ 0_K u = 0_M$
3. $\lambda(\mu u) = (\lambda\mu)u;\ 1_K u = u$

The notions of morphisms and submodules are straigthforward.

Remark 2. i) The definition above stands for left modules and we a have a similar definition for right modules.
ii) This structure amounts to the datum of a morphism of semirings $K(+,\times) \to (End(M,+),+,\circ)$.

We give now some (standard) definitions on the set of functions $X \to K$ which will be of use below.

Definition 6. *i) For any set X, the set of functions $X \to K$ is a module and will be denoted K^X. In particular, the set $K\langle\langle\Sigma\rangle\rangle := (\Sigma^*)^K$ of NFS forms a K-module.*
ii) The support of $f \in K^X$ is defined as $supp(f) = \{x \in X | f(x) \neq 0_K\}$.
iii) The subset $K^{(X)} \subset K^X$ of functions with finite support is a submodule of K^X, sometimes called the free module with basis X.

Example 3. A commutative and idempotent monoid $(M,+)$ is naturally endowed with a (unique) $\mathbb{B}$-module structure given by $1_{\mathbb{B}}x = x;\ 0_{\mathbb{B}}x = 0_M$. This setting will be used in Section 5.

Note 2. i) For implementation (as needed, for instance, after Proposition 5) an object $f \in K^{(X)}$ is better realized as a dynamic two rows array

x_1	$\cdots$	x_n
α_1	$\cdots$	α_n

$x_1 < \cdots < x_n$ being the support of f and $f(x_i) = \alpha_i$.
ii) Every module considered below will be endowed with a richer structure, that is a linear action of the free monoid on it, denoted $(?).u$ and such that $(?).(uv) = ((?).u).v$. Such a structure will be called a $K-\Sigma^*$-module structure. In fact, these actions will always come from the projections of (iterated) derivatives.

Now, we have to extend in this general framework the notion of stability mentioned in Theorem 4.

Definition 7. *i) Let $(m_i)_{i\in I}$ be a finite family in a $K-\Sigma^*$-module M. We say that it is stable by transitions (FST in the following) iff for every letter $a \in \Sigma$ and $i \in I$, we have coefficients $\mu_{ij}(a)$ such that:*

$$m_i.a = \sum_{j\in I} \mu_{ij}(a) m_j$$

Equivalently, this amounts to say that the submodule generated by the family is stable by the action of Σ^*.

ii) (λ-determinism) A FST will be called λ-deterministic if the raws of the transition matrices can be choosen with at most one non zero element. That is for every letter $a \in \Sigma$ and $i \in I$, either $m_i.a = 0_M$ or there exists $j \in I$ and $\mu_{ij}(a)$ such that:

$$m_i.a = \mu_{ij}(a)m_j$$

iii) (Determinism for a FST) A FST will be called deterministic if the raws of the transition matrices can be choosen with at most one non zero element which must be 1_K. That is for every letter $a \in \Sigma$ and $i \in I$, either $m_i.a = 0_M$ or there exists $j \in I$ such that:

$$m_i.a = m_j$$

iv) Let $\mathcal{F} = (m_i)_{i\in I}$ be a FST, then for every m linear combination of the FST (i.e. $m = \sum_{i\in I} \lambda_i m_i$) we will say that m admits the FST $\mathcal{F}$.

There is a simple criterion to test whether an element admits a deterministic FST.

Proposition 2. *(Deterministic criterion). Let M be a $K - \Sigma^*$-module. Then we have:*
i) An element $m \in M$ admits a deterministic FST iff the set $\{m.u\}_{u\in\Sigma^*}$ is finite.
ii) More precisely, if the (deterministic) FST is of cardinality n, the cardinality of the orbit of m by Σ^* (i.e. $m.\Sigma^* = \{m.u\}_{u\in\Sigma^*}$) has a cardinality which does not exceed $(n+1)^n$.

Proof. Statements i) and ii) can be proved simultaneously, considering that the cardinality of the monoid of (raw) deterministic $n \times n$ matrices (i.e. the matrices with at most one "one" on each raw) has cardinality $(n+1)^n$. □

Note 3. i) From the preceding proof one sees that, if an element admits a deterministic FST, there is a deterministic FST to which this element belongs.
ii) If m admits a FST and if K is finite, then its orbit is finite and hence, m admits a deterministic FST.
iii) The bound is reached for $\sharp\Sigma \geq 3$ and $\sharp K \geq n$.
iv)The monoid of (raw) deterministic $n \times n$ matrices (seen as mappings $f : [0..n] \to [0..n]$ such that $f(0) = 0$) is generated by:

- the transposition $(1, 2)$,
- the long cycle $(1, 2, 3 \cdots n)$,
- the projection $k \to k$; $k < n$ and $n \to 0$.

To each letter corresponds one of the preceding transitions (all of them must be choosen). Since $\sharp K \geq n$ we can take a family of n different coefficients $(\lambda_1, \lambda_2, \cdots \lambda_n)$. Using the standard process to compute a FST with given transition matrices, we see that the expression with coordinate vector $(\lambda_1, \lambda_2, \cdots \lambda_n)$ has an orbit with exactly $(n+1)^n$ elements.

The characterization for the λ-determinism is not so simple. It is possible, however, to complete it in the case of one variable ($\Sigma = \{a\}$) and K a (commutative) field.

Proposition 3. *(λ-deterministic criterion). Let $\Sigma = \{a\}$ be a one letter alphabet and K be a field. Let M be a $K - \Sigma^*$-module. Then, an element $m \in M$ admits a λ-deterministic FST iff there exists an $N \in \mathbb{N} - \{0\}$ such that the module generated by $(m.a^n)_{n \geq N}$ is finite dimensional and if a^N acts diagonally on it.*

4.2 Admissible Congruences: A Basic List

Now, we want to compute with rational expressions, so we need to give us additional rules. These rules must preserve the actions of $(a^{-1}(?))_{a \in \Sigma}$ and, since they must describe rational series, they must be finer than $\sim_{rat}$.

Definition 8. *i) A congruence $\sim$ on the set $\mathcal{E}(K, \Sigma)$ will be called admissible iff it is finer than $\sim_{rat}$ and compatible with the operators a^{-1} and the const mapping.*
ii) We give the following list of congruences on $\mathcal{E}(K, \Sigma)$:
- $E_1 + (E_2 + E_3) \sim (E_1 + E_2) + E_3$ *(A+)*
- $E_1 + E_2 \sim E_2 + E_1$ *(C)*
- $E + 0_{\mathcal{E}} \sim 0_{\mathcal{E}} + E \sim E$ *(N)*
- $\lambda(E + F) \sim \lambda E + \lambda F;\ \lambda 0_{\mathcal{E}} \sim 0_{\mathcal{E}}$ *($ExtD_l$)*
- $(\lambda + \mu)E \sim \lambda E + \mu E;\ 0_K E \sim 0_{\mathcal{E}}$ *($ExtD_r$)*
- $\lambda(\mu E) \sim (\lambda\mu)E;\ 1_K E \sim E$ *(ExtA)*
- $(E + F) \cdot G \sim E \cdot G + F \cdot G;\ 0_{\mathcal{E}} \cdot F \sim 0_{\mathcal{E}}$ *(D_r)*
- $E \cdot (F + G) \sim E \cdot F + E \cdot G;\ E \cdot 0_{\mathcal{E}} \sim 0_{\mathcal{E}}$ *(D_l)*
- $\varepsilon \cdot E \sim E$ *(Ul) (Unit left)*
- $E \cdot \varepsilon \sim E$ *(Ur) (Unit right)*
- $(\lambda E) \cdot F \sim \lambda(E \cdot F)$ *(MixA·)*
- $E \cdot (F \cdot G) \sim (E \cdot F) \cdot G$ *(A·)*
- $E^* \sim \varepsilon + E \cdot E^*$ *(Star)*

iii) The $\sim_{acm1}$ congruence is defined by [A+,C,N,Ext(D_l,D_r,A)].
$\sim_{acm2}$ is defined by $\sim_{acm1} \wedge MixA\cdot \wedge D_r$ that is [A+,C,N,MixA· ,Ext(D_l,D_r,A)].
$\sim_{acm3}$ is defined by $\sim_{acm1} \wedge MixA\cdot, \wedge A\cdot, \wedge D_{r,l} \wedge U_{r,l}$ that is [A+,C,N,MixA·,A·, D_r,D_l,U_r,U_l,Ext(D_l,D_r,A)].
iv) In the following $\mathcal{E}/\sim_{acmi}$ will be denoted $\mathcal{E}_i$.

Proposition 4. *i) The set of admissible congruences is a complete sublattice of the lattice of all congruences on $\mathcal{E}(K, \Sigma)$.*
ii) All the $\sim_{acmi}$ are admissible congruences.

Remark 3. i) Of course, one has $\sim_{acm1} \subset \sim_{acm2} \subset \sim_{acm3}$.
ii) The congruence $\sim_{acm1}$ is the finest one such that the quotients of the laws (sum and external product of $\mathcal{E}(K, \Sigma)$) endow the quotient $\mathcal{E}/\sim$ with a K-module structure.
iii) For every admissible congruence $\sim$ coarser than $\sim_{acm1}$, the quotient $\mathcal{E}/\sim$ is canonically endowed with a (left) K-module structure (and hence a $K - \Sigma^*$-module structure since it is a^{-1}-compatible).

The following proposition states that there is a tractable normal form in every quotient $\mathcal{E}_i = \mathcal{E}/\sim_{acmi}$, for $i = 1, 2, 3$.

Theorem 5. *The modules $\mathcal{E}_i$; $i = 1, 2, 3$ are free.*

4.3 An Analogue for a Theorem of Antimirov

Now, we state an analogue of a theorem of Antimirov in our setting.

Theorem 6. *i) To every (class of) rational expression(s) $E \in \mathcal{E}/\sim_{acm2}$, one can associate algorithmically a FST $\mathcal{F}_E = (E_i)_{i\in I}$ such that E is a linear combination of $\mathcal{F}_E$.*
ii) (Deterministic property) If the semiring is finite, then the set of derivatives in $\mathcal{E}/\sim_{acm1}$ of every rational expression is finite and hence admits a deterministic FST.

Remark 4. The algorithms provided by the step by step construction are not always the best possible (see [13] for a probabilistic discussion on this point). One could, when it happends, avoid redundancy; see below an example where this can be done.

Example 4. Let $E = x^*(xx+y)^*$. The following FST's are inductively computed:

$$\begin{aligned}
fst_x &= \{x, \varepsilon\}\\
fst_{x^*} &= \{xx^*, \varepsilon\}\\
fst_{xx} &= \{xx, x, \varepsilon\}\\
fst_{xx+y} &= \{xx, x, y, \varepsilon\}\\
fst_{(xx+y)^*} &= \{xx(xx+y)^*, x(xx+y)^*, y(xx+y)^*, \varepsilon\}\\
fst_{E=x^*(xx+y)^*} &= \{E_1 = xx^*(xx+y)^*, E_2 = xx(xx+y)^*, E_3 = x(xx+y)^*,\\
&\qquad E_4 = y(xx+y)^*, E_5 = \varepsilon\}\\
E &= E_1 + E_2 + E_4 + E_5
\end{aligned}$$

The previous theorem predicts the existence of a (algorithmically constructible) FST in the generated submodule of which every term is embedded. If K is a field or a finite semiring one can take a finite set of derivatives. This is not possible in general as shown by the following critical counterexample.

Example 5. Let $K = \mathbb{N}$ and $E = a^* \cdot a^*$. Then, applying the rules, one can show that, in $\mathcal{E}/\sim_{acm3}$, we have $a^{-n}E = E + na^*$ and so, the set of derivatives of E is infinite and cannot be generated by any finite subset of it. Moreover, the associated series admits no deterministic recognizer and hence it is so for E itself.

In fact, looking closer to the proof of Theorem 6 (ii), one sees that the conclusion holds if the semiring verifies the following weaker property:

Property B [1]**:** *The submonoid generated by a finite number of matrices in* $K^{n\times n}$ *is finite.*

Note 4. It is clear that finiteness implies Property B but the converse is false as shown by the semiring $\mathbf{B}^{(\mathbf{N})} \oplus \mathbf{B}.\mathbf{1}_{\mathbf{N}}$, the subsemiring of functions $\mathbf{N} \to \mathbf{B}$ being either almost everywhere 0 or almost everywhere 1.

5 Determinism and the Converse of a Theorem of Brzozowski

Our concern here is to study the existence of deterministic recognizers. We give a generalization of Brzozowski's theorem and its converse in the sense that we provide a necessary and sufficient condition over the semiring K so that every automaton could have a deterministic counterpart. Now, we weaken the $\sim_{acm1}$ equivalence so that, by specialization to $K = \mathbf{B}$ one should recover $\sim_{aci}$.

Definition 9. *For a semiring* K*, the* $\sim_{acs}$ *equivalence is defined, on the set* $\mathcal{E}_0 = \mathcal{E}(K, \Sigma) \cup \{\omega\}$ *(*$\omega \notin \mathcal{E}$*) by the pairs*

- $E_1 + (E_2 + E_3) \sim (E_1 + E_2) + E_3$ *(A+)*
- $E_1 + E_2 \sim E_2 + E_1$ *(C)*
- $E + \omega \sim \omega + E \sim E$ *(N)*

and the (S) relations

- $\lambda(E + F) \sim \lambda E + \lambda F;\ \lambda\omega \sim \omega$ *(ExtDl)*
- $(\lambda + \mu)E \sim \lambda E + \mu E;\ 0_K E \sim \omega$ *(ExtDr)*
- $\lambda(\mu E) \sim (\lambda\mu)E;\ 1_K E \sim E$ *(ExtA)*

One extends the operators a^{-1} to $\mathcal{E}_0$ by $a^{-1}(\omega) = \omega$. Then it is easy to check that $E \sim_{acs} F \Longrightarrow a^{-1}(E) \sim_{acs} a^{-1}(F)$.

Remark 5. One can check, in view of Example 3, that the trace on $\mathcal{E}$ of the congruence $\sim_{acs}$, in case $K = \mathbf{B}$, is the $\sim_{aci}$ congruence of Brzozowski.

Theorem 7. *For any semiring, the following conditions are equivalent:*
(i) For every $E \in \mathcal{E}_0/\sim_{acs}$*, the set* $\{u^{-1}E\}_{u\in\Sigma^*}$ *is finite.*
(ii) K *satisfies property* B .

5.1 Reconstruction Lemma, Congruence $\sim_{acm3}$ and the Linear Forms of Antimirov

A well known lemma in language theory (and a little less in the theory of series) states that, for a series $S \in K\langle\langle\Sigma\rangle\rangle$ and with $const(S) =< S|\varepsilon >$, one has:

$$S = const(S)\varepsilon + \sum_{a\in\Sigma} a\ (a^{-1}S)$$

[1] In honour of Burnside, Brzozowski and Boole. Note that condition B is stronger that Burnside condition [10] for semirings.

This equality can be stated (but, of course not necessarily satisfied) in $\mathcal{E}(K,\Sigma)/\sim$ for all admissible congruence which satisfies (A+) and (C). We will call it the *reconstruction lemma* or, for short, (RL) [15]. We establish the equivalence of (RL) and (Star) ($E^* \sim \varepsilon + E \cdot E^*$). Otherwise stated, if one of these two statement holds, the other does.

Theorem 8. *Let $\sim$ be an admissible congruence coarser than $\sim_{acm3}$. Then (Star) and (RL) are equivalent within $\mathcal{E}/\sim$.*

6 Conclusion

We have studied several congruences; our results can be summarized as follows:

	$\sim_{acm1}$	$\sim_{acm2}$	$\sim_{acm3}$
Feature	$K - \Sigma^*$-module structure Determinism (K of type B)	FST (existence)	Reconstruction Lemma $\Leftrightarrow$ Star

References

[1] V. Antimirov, *Partial derivatives of regular expressions and finite automaton constructions*, Theoretical Computer Science, **155**, 291-319 (1996). 52, 53, 57

[2] J. Berstel and D. Perrin *Theory of codes*, Academic Press (1985).

[3] J. Berstel and C. Reutenauer, *Rational Series and Their Languages* (EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1988). 52, 54

[4] J. A. Brzozowski. Derivatives of regular expressions. *J. Assoc. Comput. Mach.*, 11(4):481–494, 1964. 52, 53, 57

[5] J. H. Conway, *Regular Algebras and Finite Machines*, Chapman and Hall, London 1974. 52, 57

[6] K. Culik II and J. Kari, *Finite state transformations of images*, Proceedings of ICALP 95, Lecture Notes in Comput. Sci. **944** (1995) 51-62.

[7] J.-M. Champarnaud and D. Ziadi, New Finite Automaton Constructions Based on Canonical Derivatives, in CIAA'2000, *Lecture Notes in Computer Science*, S. Yu ed., Springer-Verlag, to appear. 52

[8] J.-M. Champarnaud and D. Ziadi, From Mirkin's Prebases to Antimirov's Word Partial Derivatives, *Fundamenta Informaticae*, 45(3), 195–205, 2001. 52

[9] J.-M. Champarnaud and D. Ziadi, Canonical Derivatives, Partial Derivatives, and Finite Automaton Constructions, *Theoret. Comp. Sc.*, to appear. 52

[10] M. Droste, P. Gastin, *On Aperiodic and Star-free Formal Power Series in partially Commuting variables*, Proceedings of FPSAC'00, D. Krob, A. A. Mikhalev and A. V. Mikhalev. (Springer, june 2000). 62

[11] G. Duchamp, D. Krob, *Combinatorics on traces*, Ch II of the "Book of traces" EATCS monography. (1995) (Ed. G. Rozenberg, V. Dieckert) World Scientific. 55

[12] G. Duchamp and C. Reutenauer, Un critère de rationalité provenant de la géométrie non-commutative, *Invent. Math.* **128** (1997) 613–622. 52

[13] G. Duchamp, M. Flouret, É. Laugerotte, J.-G. Luque, Direct and dual laws for automata with multiplicities, *Theoret. Comp. Sc.*, 269/1-2, to appear. 61

[14] S. Eilenberg, *Automata, languages and machines, Vol. A* (Acad. Press, New-York, 1974). 54, 55
[15] G. Jacob, Représentations et substitutions matricielles dans la théorie algébrique des transductions. Thèse d'état. Université Paris VII (1975). 63
[16] S. C. Kleene, Representation of events in nerve nets and finite automata, *Automata Studies, Princeton Univ. Press* (1956) 3–42.
[17] D. Krob, *Models of a K-rational identity system*, Journal of Computer and System Sciences, **45**, (3), 396-434, 1992. 52, 57
[18] D. Krob, *Differentiation of K-rational expressions identity system*, International Journal of Algebra and Computation, **3** (1), 15-41, 1993.
[19] M. Lothaire, *Combinatorics on words* (Addison-Wesley, 1983).
[20] S. Mac Lane *Categories for the Working Mathematician*, Springer (4th ed. 1988). 56
[21] B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.
[22] M. Mohri, F. Pereira and M. Riley, A Rational Design for a Weighted Finite-State Transducer Library. *Lecture Notes in Computer Science*, 1436:43–53 , 1998.
[23] C. Reutenauer, *A survey on noncommutative rational series*, FPSAC'94 proceedings. 52
[24] A. Salomaa and M. Soittola, *Automata-theoretic aspects of formal power series.* (Springer-Verlag, 1978).
[25] M. P. Schützenberger, *On the definition of a family of automata*, Information and Control **4** (1961) 245–270. 52
[26] R. P. Stanley, *Enumerative combinatorics*, Vol 2, Cambridge (1999). 54
[27] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume I, Words, Languages, Grammars, pages 41–110. Springer-Verlag, Berlin, 1997. 53

Finite Automata for Compact Representation of Language Models in NLP

Jan Daciuk and Gertjan van Noord

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

Abstract. A technique for compact representation of *language models* in Natural Language Processing is presented. After a brief review of the motivations for a more compact representation of such *language models*, it is shown how finite-state automata can be used to compactly represent such language models. The technique can be seen as an application and extension of *perfect hashing* by means of finite-state automata. Preliminary practical experiments indicate that the technique yields considerable and important space savings of up to 90% in practice.

1 Introduction

An important practical problem in Natural Language Processing (NLP) is posed by the size of the knowledge sources that are being employed. For NLP systems which aim at full parsing of unrestricted texts, for example, realistic electronic dictionaries must contain information for hundreds of thousands of words. In recent years, *perfect hashing* techniques have been developed based on finite state automata which enable a very compact representation of such large dictionaries without sacrificing the time required to access the dictionaries [7, 11, 10]. A freely available implementation of such techniques is provided by one of us [4, 3][1].

A recent experience in the context of the Alpino wide-coverage grammar for Dutch [1] has once again established the importance of such techniques. The Alpino lexicon is derived from existing lexical resources. It contains almost 50,000 stems which give rise to about 200,000 fully inflected entries in the compiled dictionary which is used at runtime. Using a standard representation provided by the underlying programming language (in this case Prolog), the lexicon took up about 27 Megabytes. A library has been constructed (mostly implemented in C++) which interfaces Prolog and C with the tools provided by the `s_fsa` [4, 3] package. The dictionary now contains only 1,3 Megabytes, without a noticeable delay in lexical lookup times.

However, dictionaries are not the only space consuming resources that are required by current state-of-the-art NLP systems. In particular, *language models* containing statistical information about the Co-occurrence of words and/or word

[1] http://www.pg.gda.pl/~jandac/fsa.html

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 65–73, 2002.
© Springer-Verlag Berlin Heidelberg 2002

meanings typically require even more space. In order to illustrate this point, consider the model described in chapter 6 of [2]; a recent, influential, dissertation in NLP. That chapter describes a statistical parser which bases its parsing decisions on bigram lexical dependencies, trained from the Penn Treebank. Collins reports:

> All tests were made on a Sun SPARCServer 1000E, using 100% of a 60Mhz SuperSPARC processor. The parser uses around 180 megabytes of memory, and training on 40,000 sentences (essentially extracting the co-occurrence counts from the corpus) takes under 15 minutes. Loading the hash table of bigram counts into memory takes approximately 8 minutes.

A similar example is described in [5]. Foster compares a number of linear models and maximum entropy models for parsing, considering up to 35,000,000 features, where each feature represents the occurrence of a particular pair of words.

The use of such data-intensive probabilistic models is not limited to parsing. For instance, [8] describes a method to learn the ordering of prenominal adjectives in English (from the British National Corpus), for the purpose of a natural language generation system. The resulting model contains counts for 127,016 different pairs of adjectives.

In practice, systems need to be capable to work not only with bigram models, but trigram and fourgram models are being considered too. For instance, an unsupervised method to solve PP-attachment ambiguities is described in [9]. That method constructs a model, based on a 125-million word newspaper corpus, which contains counts of the relevant $\langle V, P, N_2 \rangle$ and $\langle N_1, P, N_2 \rangle$ trigrams, where P is the preposition, V is the head of the verb phrase, N_1 is the head of the noun phrase preceding the preposition, and N_2 is the head of the noun phrase following the preposition. In speech recognition, language models based on trigrams are now very common [6].

For further illustration, a (Dutch) newspaper corpus of 40,000 sentences contains about 60,000 word types; 325,000 bigram types and 530,000 trigram types. In addition, in order to improve the accuracy of such models, much larger text collections are needed for training. In one of our own experiments we employed a Dutch newspaper corpus of about 350,000 sentences. This corpus contains more than 215,000 unigram types, 1,785,000 bigram types and 3,810,000 trigram types. A straightforward, textual, representation of the trigram counts for this corpus takes more than 82 Megabytes of storage. Using a standard hash implementation (as provided by the `gnu` version of the C++ standard library), will take up 362 Megabytes of storage during run-time. Initializing the hash from the table takes almost three minutes. Using the technique introduced below, the size is reduced to 49 Megabytes; loading the (off-line constructed) compact language model takes less than half a second.

All the examples illustrate that the size of the knowledge sources that are being employed is an important practical problem in NLP. The runtime memory

requirements become problematic, as well as the CPU-time required to load the required knowledge sources. In this paper we propose a method to represent huge language models in a compact way, using finite-state techniques. Loading compact models is much faster, and in practice no delay in using these compact models is observed.

2 Formal Preliminaries

In this paper we attempt to generalize over the details of specific statistical models that are employed in NLP systems. Rather, we will assume that such models are composed of various functions from tuples of strings to tuples of numbers. Each such *language model function* $T^{i,j}$ is a finite function $(W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots \times Z_j)$. The word columns typically contain words, word meanings, the names of dependency relations, part-of-speech tags and so on. The number columns typically contain counts, the cologarithm of probabilities, or other numerical information such as *diversity*.

For a given language model function $T^{i,j}$, it is quite typical that some of the dictionaries $W_1 \ldots W_i$ may in fact be the same dictionary. For instance, in a table of bigram counts, the set of first words is the same as the set of second words. The technique introduced below will be able to take advantage of such shared dictionaries, but does not require that the dictionaries for different columns are the same. Naturally, more space savings can be expected in the first case.

3 Compact Representation of Language Models

A given language model function $T^{i,j} : (W_1 \times \ldots \times W_i) \rightarrow (Z_1 \times \ldots \times Z_j)$ is represented by (at most) i perfect hash finite automata, as well as a table with $i+j$ rows. Thus, for each W_k, we construct an acyclic finite automaton out of all words found in W_k. Such an automaton has additional information compiled in, so that it implements perfect hashing ([7],[11],[10]). The perfect hash automaton (fig. 1) converts between a word $w \in W_k$ and a unique number $0 \leq |W_k| - 1$. We write $N(w)$ to refer to the *hash key* assigned to w by the corresponding perfect hash automaton.

If there is enough overlap between words from different columns, then we might prefer to use the same perfect hash automaton for those columns. This is a common situation in n-grams used in statistical natural language processing.

We construct a table such that for each $w_1 \ldots w_i$ in the domain of T, where $T(w_1 \ldots w_i) = (z_1 \ldots z_j)$, there is a row in the table consisting of $N(w_1), \ldots, N(w_i)$, $z_1, \ldots, z_j$. Note that all cells in the table contain numbers. We represent each such number on as few bytes as are required for the largest number in its column. The representation is not only compact (a number is typically represented on 2 instead of 8 bytes on a 64 bit architecture), but it is machine-independent (in our implementation, the least significant byte always comes first). The table is sorted. So a language model function is represented

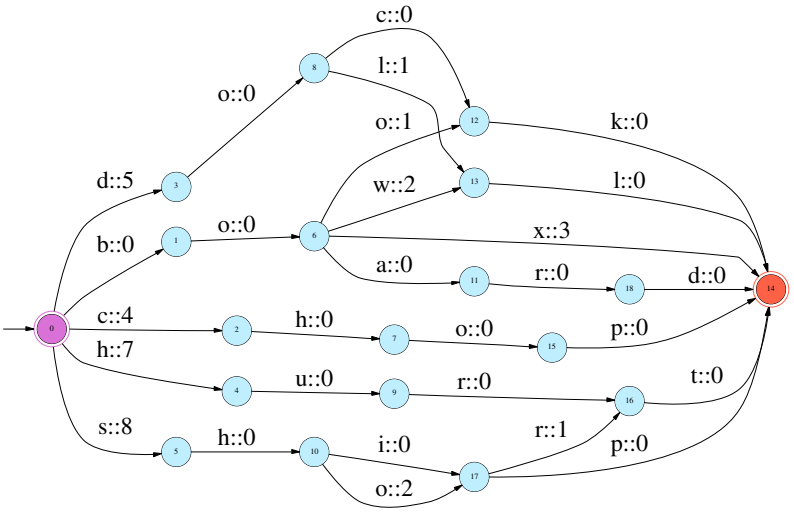


Fig. 1. Example of a perfect hash automaton. The sum of numbers along transitions recognizing a given word give the word number (hash key). For example, *doll* has number 5+0+1+0=6

by a table of packed numbers, and at most i perfect hash automata converting words into the corresponding hash keys.

The access to a value $T(w_1 \dots w_n)$ involves converting the words $w_1 \dots w_n$ to their hash keys $N(w_1) \dots N(w_n)$ using perfect hashing automata; constructing a query string from the hash keys by packing these hash keys; and using a binary search for the query string in the table; $T(w_1 \dots w_n)$ is then obtained by unpacking the values found in the table.

There is a special case for language model functions $T^{i,j}$ where $i = 1$. Because the words are unique, their hash keys are unique numbers form $0 \dots |W_1| - 1$, and there is no need to store the hash key of the words in the table. The hash key just serves as an index in the table. Also the access is different than in the general case. After we obtain the hash key, we use it as the address of the numerical tuple.

4 Preliminary Results

We have performed a number of preliminary experiments. The results are summarized in table 1. The *text* method indicates the size required by a straightforward textual representation. The *old* methods indicate the size required for a straightforward Prolog implementation (as a long list of facts) and a standard implementation of hashes in C++. It should be noted that a hash would always require at least as much space as the *text* representation. We compared

our method with the hash-map datastructure provided by the `gnu` implementation of the C++ standard library (this was the original implementation of the knowledge sources in the bigram POS-tagger, referred to in the table).[2]

The *concat dict* method indicates the size required if we treat the sequences of strings as words from a single dictionary, which we then represent by means of a finite automaton. No great space savings are achieved in this case (except for the Alpino tuple) , because the finite automaton representation is able only to compress prefixes and suffixes of words; if these 'words' get very long (as you get by concatenating multiple words) then the automaton representation is not suitable. The final *new* column indicates the space required by the new method introduced in this paper.

We have compared the different methods on various inputs. The *Alpino tuple* contains tuples of two words, two part-of-speech tags, and the name of a dependency relation. It relates such a 5-tuple with a tuple consisting of three numbers. The rows labeled *n sents trigram* refer to a test in which we calculated the trigram counts for a Dutch newspaper corpus of *n* sentences. The *n sents fourgram* rows are similar, but this case we computed the fourgram counts. Because all words in n-gram tests came from the same dictionary, we needed only one automaton instead of 3 for trigrams and 4 for fourgrams. The automaton sizes for trigrams accounted for 11.84% (20 000 sentences) and 9.33% (40 000 sentences) of the whole new representation, for fourgrams – 8.59% and 6.53% respectively. The automata for the same input data size were almost identical. Finally, the *POS-tagger* row presents the results for an HMM part-of-speech tagger for Dutch (using a tag set containing 8,644 tags), trained on a corpus of 232,000 sentences. Its knowledge sources are a table of bigrams of tags (containing 124,209 entries) and a table of word/tag pairs (containing 209,047 entries).

As can be concluded from the results in table 1, the new representation is in all cases the most compact one, and generally uses less than half of the space required by the textual format. Hashes, which are mostly used in practice for this purpose, consistently require about ten times as much space.

5 Variations and Future Work

We have investigated additional methods to compress and speed-up the representation and use of language model functions; some other variations are mentioned here as pointers to future work.

In the table, the hash key in the first column can be the same for many rows. For trigrams, for example, the first two hash keys may be identical for many rows of the table. In the trigram data set for 20,000 sentences, 47 rows (out of 295,303) have hash key 1024 in the first column, 10 have 0, 233 – 7680. The same situation can arise for other columns. In the same data set, 5 rows have 1024 in

[2] The sizes reported in the table are obtained using the Unix command `wc -c`, except for the size of the hash. Since we did not store these hashes on disk, the sizes were estimated from the increase of the memory size reported by `top`. All results are obtained on a 64bit architecture.

Table 1. Comparison of various representations (in Kbytes)

test set	text	old		concat	dict	new
		Prolog	C++ hash			
Alpino tuple	9,475	44,872	NA	4,636		4,153
20,000 sents trigram	5,841	32,686	27,000	6,399		2,680
40,000 sents trigram	11,320	61,672	52,000	11,113		4,975
20,000 sents fourgram	8,485	45,185	33,000	13,659		3,693
40,000 sents fourgram	16,845	88,033	65,000	20,532		7,105
POS-tagger	15,722	NA	45,000	NA		4,409

the first column, and 29052 in the second column, 16 – 7680 in the first column, and 17359 in the second one. By representing them once, and providing a pointer to the remaining part, and doing the same recursively for all columns, we arrive at a structure called *trie*. In the trie, edges going out from root are labeled with all the hash keys from the first column. They point to vertices with outgoing edges representing tuples that have the same two words at the beginning, and so on. By keeping only one copy of hash keys from the first few columns, we hope to economize the storage space. However, we also need additional memory for pointers. A vertex is represented as a vector of edges, and each edge consists of two items: the label (hash key), and a pointer. The method works best when the table is dense, and when it has very few columns. We construct the trie only for the columns representing words; we keep the numerical columns intact (obviously, because it is "output").

For dense tables, we may perceive the trie as a finite automaton. The vertices are states, and the edges – transitions. We can reduce the number of states and transition in the automaton by minimizing it. In that process, isomorphic subtrees of the automaton for the word columns are replaced with single copies. This means that additional sharing of space takes place. However, we need to determine which paths in the automaton lead to which sequences of numbers in

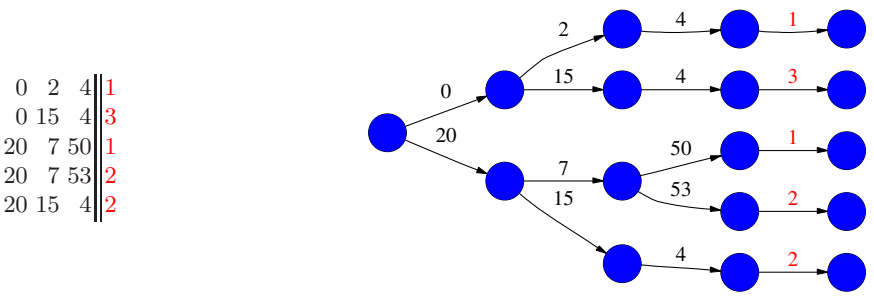


Fig. 2. Trie (right) representing a table (left). Red labels represent numerical tuples. Numbers 0 and 20 from the first column, and 7 from the second column, are represented only once

0	2	4	1
0	15	4	3
20	7	50	1
20	7	53	2
20	15	4	2

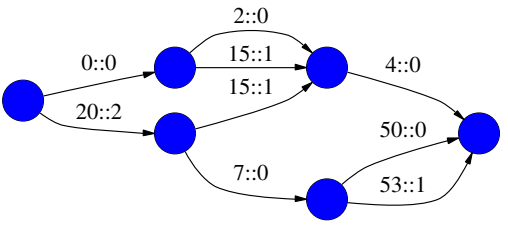


Fig. 3. Perfect hash automaton (right) representing a table (left). Only word columns are represented in the automaton. Numerical columns from the table are left intact. They are indexed by hash keys (sums of numbers after "::" on transitions). The first row has index 0

the numerical columns. This is done, again, by means of *perfect hashing*. This implies that each transition in the automaton not only contains a label (hash key) and a pointer to the next state, but also a number which is required to construct the hash key. Although we share more transitions, we need space for storing those additional numbers.

We use a sparse matrix representation to store the resulting minimal automaton. The look-up time in the table for the basic model described in the previous section is determined by binary search. Therefore, the time to look-up a tuple is proportional to the binary logarithm of the number of tuples. It may be possible to improve on the access times by using interpolated search instead of binary search. In an automaton, it is possible to make the look-up time independent from the number of tuples. This is done by using the sparse matrix representation ([12]) applied to finite-state automata ([10]). A state is represented as a set of transitions in a big vector of transitions for the whole automaton. We have a separate vector for every column. This allows us to adjust the space taken by pointers and numbering information. The transitions do not have to occupy adjacent space; they are indexed with their labels, i.e. the label is the transition number. As there are gaps between labels, there are also gaps in the representation of a single state. They can be filled with transitions belonging to other states, provided that those states do not begin at the same point in the transition vector. However, it is not always possible to fill all the gaps, so some space is wasted.

Results on the representation of language model functions using minimal automata for word tuples and sparse matrix representation are discouraging. If we take the word tuples, and create an automaton with each row converted to a string of transitions labeled with hash keys from successive columns, and then minimize that automaton, and compare the number of transitions, we get from 27% to 44% reduction. However, the transition holds two additional items, usually of the same size as the label, which means that it is 3 times as big as a simple label. In the trie representation, we don't need numbering information, so the transition is twice as big as the label, but the automaton has even more

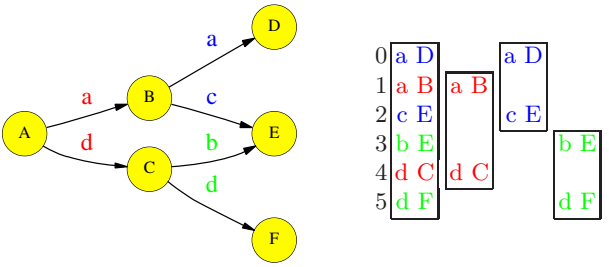


Fig. 4. Sparse table representation (right) of a part of an automaton (left). Node A has number 1, B – 0, C – 3. The first column is the final representation, column 2 – state A, column 3 – state B, column 4 – state C

transitions. Also, the sparse matrix representation introduces additional loss of space. In the our experiments, 32% to 59% of space in the transition vector is not filled. This loss is due to the fact that the labels on outgoing transitions of a state can be any subset of numbers from 0 to over 50,000. This is in sharp contrast with natural language dictionaries, for instance, where the size of the alphabet is much smaller. We also tried to divide longer (i.e. more than 1 byte long) labels into a sequence of 1 byte long labels. While that led to better use of space and more transition sharing, it also introduced new transitions, and the change in size was not significant. The sparse matrix representation was in any case up to 3.6 times bigger than the basic one (table of hash keys), with only minor improvement in speed (up to 5%).

We thought of another solution, which we did not implement. We could represent a language model function $T^{i,j}$ as an i-dimensional array $A[1, \ldots, i]$. As before, there are perfect hashing automata for each of the dictionaries $W_1 \ldots W_n$. For a given query $w_1 \ldots w_n$, the value $[N(w_1), \ldots, N(w_n)]$ is then used as an index into the array A. Because the array is typically very sparse, it should be stored using a sparse matrix representation. It should be noted that this approach would give very fast access, but the space required to represent A is at least as big (depending on the success of the sparse matrix representation) as the size of the table constructed in the previous method.

6 Conclusions

We have presented a new technique for compact representation of language models in natural language processing. Although it is a direct application of existing technology, it has great practical importance (numerous examples are quoted in the introduction), and we have demonstrated that our solution is the answer to the problem. We also show that a number of more sophisticated and scientifically appealing techniques are actually inferior to the basic method presented in the paper.

Acknowledgments

This research was carried out within the framework of the PIONIER Project *Algorithms for Linguistic Processing*, funded by NWO (Dutch Organization for Scientific Research) and the University of Groningen.

References

[1] Gosse Bouma, Gertjan van Noord, and Robert Malouf. Wide coverage computational analysis of Dutch. 2001. Submitted to volume based on CLIN-2000. Available from `http://www.let.rug.nl/~vannoord/`. 65

[2] Michael Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University Of Pennsylvania, 1999. 66

[3] Jan Daciuk. Experiments with automata compression. In M. Daley, M. G. Eramian, and S. Yu, editors, *Conference on Implementation and Application of Automata CIAA'2000*, pages 113–119, London, Ontario, Canada, July 2000. University of Western Ontario. 65

[4] Jan Daciuk. Finite-state tools for natural language processing. In *COLING 2000 Workshop on Using Tools and Architectures to Build NLP Systems*, pages 34–37, Luxembourg, August 2000. 65

[5] George Foster. A maximum entropy/minimum divergence translation model. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 37–44, Hong Kong, October 2000. 66

[6] Frederick Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, 1998. 66

[7] Claudio Lucchiesi and Tomasz Kowaltowski. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, Jan. 1993. 65, 67

[8] Robert Malouf. The order of prenominal adjectives in natural language generation. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 85–92, Hong Kong, October 2000. 66

[9] Patrick Pantel and Dekang Lin. An unsupervised approach to prepositional phrase attachment using contextually similar words. In K. Vijay-Shanker and Chang-Ning Huang, editors, *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 101–108, Hong Kong, October 2000. 66

[10] Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes*. PhD thesis, Institut Blaise Pascal, Paris, France, 1991. LITP 91.44. 65, 67, 71

[11] Emmanuel Roche. Finite-state tools for language processing. In *ACL'95*. Association for Computational Linguistics, 1995. Tutorial. 65, 67

[12] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979. 71

Past Pushdown Timed Automata
(Extended Abstract)

Zhe Dang[1], Tevfik Bultan[2], Oscar H. Ibarra[2], and Richard A. Kemmerer[2]

[1] School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164
[2] Department of Computer Science
University of California, Santa Barbara, CA 93106

Abstract. We consider past pushdown timed automata that are discrete pushdown timed automata [15] with past-formulas as enabling conditions. Using past formulas allows a past pushdown timed automaton to access the past values of the finite state variables in the automaton. We prove that the reachability (i.e., the set of reachable configurations from an initial configuration) of a past pushdown timed automaton can be accepted by a nondeterministic reversal-bounded multicounter machine augmented with a pushdown stack (i.e., a reversal-bounded NPCM). Using the known fact that the emptiness problem for reversal-bounded NPCMs is decidable, we show that model-checking past pushdown timed automata against Presburger safety properties on discrete clocks and stack word counts is decidable. An example ASTRAL specification is presented to demonstrate the usefulness of the results.

1 Introduction

As far as model-checking is concerned, the most successful model of infinite state systems that has been investigated is probably timed automata [2]. A timed automaton can be considered as a finite automaton augmented with a number of clocks. Enabling conditions in a timed automaton are in the form of (clock) *regions*: a clock or the difference of two clocks is tested against an integer constant, e.g., $x - y < 8$. The region technique [2] has been used to analyze region reachability, to develop a number of temporal logics [1, 3, 4, 5, 20, 24, 26, 29] and for model-checking tools [19, 23, 30].

The region technique is useful, but obviously not enough. For instance, it is not possible, using the region technique, to verify whether clock values satisfying a non-region property $x_1 - x_2 > x_3 - x_4$ are reachable for a timed automaton. The verification calls for a decidable characterization for the binary reachability (all the configuration pairs such that one can reach the other) of a timed automaton. The characterizations have recently been established in [9] for timed automata and in [15, 12] for timed automata augmented with a pushdown stack.

In this paper, we consider a class of discrete timed systems, called past pushdown timed automata. In a past pushdown timed automaton, the enabling condition of a transition can access some finite state variable's past values. For

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 74–86, 2002.
© Springer-Verlag Berlin Heidelberg 2002

instance, consider discrete clocks x_1, x_2 and now (a clock that never resets, indicating the current time). Suppose that a and b are two Boolean variables. An enabling condition could be in the form of a past formula:

$\forall 0 \leq y_1 \leq now \exists 0 \leq y_2 \leq now((x_1 - y_1 < 5 \wedge a(x_1) = b(y_2)) \rightarrow (y_2 < x_2 + 4))$, in which $a(x_1)$ and $b(y_2)$ are (past) values of a and b at times x_1 and y_2, respectively. Thus, past pushdown timed automata are history dependent; that is, the current state depends upon the entire history of the transitions leading to the state. The main result of this paper shows that the reachability of past pushdown timed automata can be accepted by reversal-bounded multicounter machines augmented with a pushdown stack (i.e., reversal-bounded NPCMs). Since the emptiness problem for reversal-bounded NPCMs is decidable [21], we can show that checking past pushdown timed automata against Presburger safety properties on discrete clocks and stack word counts is decidable. This result is not covered by region-based results for model-checking timed pushdown systems [7], nor by model-checking pushdown systems [6, 16].

Besides their own theoretical interest, history-dependent timed systems have practical applications. It is a well known principle that breaking a system into several loosely independent functional modules greatly eases both verification and design work. The ultimate goal of modularization is to partition a large system, both conceptually and functionally, into several small modules and to verify each small module instead of verifying the large system as a whole. That is, verify the correctness of each module without looking at the behaviors of the other modules. This idea is adopted in a real-time specification language ASTRAL [8], in which a module (called a process) is provided with an interface section, which is a first-order formula that abstracts its environment. It is not unusual for these formulas to include complex timing requirements that reflect the patterns of variable changes. Thus, in this way, even a history independent system can be specified as a number of history dependent modules (see [8, 10, 13] for a number of interesting real-time systems specified in ASTRAL). Thus, the results in this paper have immediate applications in implementing an ASTRAL symbolic model checker.

Past formulas are not new. In fact, they can be expressed in TPTL [5], which is obtained by including clock constraints (in the form of clock regions) and freeze quantifiers in the Linear Temporal Logic (LTL) [25]. But, in this paper, we put a past formula into the enabling condition of a transition in a generalized timed system. This makes it possible to model a real-time machine that is history-dependent. Past formulas can be expressed through S1S (see Thomas [28] and Straubing [27] for details), which can be characterized by Buchi (finite) automata. This fact does not imply (at least not in an obvious way) that timed automata augmented with these past formulas can be simulated by finite automata. In this extended abstract, the complete ASTRAL specification, as well as some substantial proofs, in particular for Theorems 3, 4 and 6 are omitted. For a complete exposition see [11].

2 Preliminaries

A *nondeterministic multicounter machine (NCM)* is a nondeterministic machine with a finite set of *(control) states* Q, and a finite number of counters with integer counter values. Each counter can add 1, subtract 1, or stay unchanged. These counter assignments are called *standard assignments*. The machine can also test whether a counter is equal to, greater than, or less than an integer constant, and these tests are called *standard tests*.

An NCM can be augmented with a pushdown stack. A *nondeterministic pushdown multicounter machine (NPCM)* M is a nondeterministic machine with a finite set of *(control) states* Q, a pushdown stack with stack alphabet Π, and a finite number of counters with integer counter values. Both assignments and tests in M are standard. In addition, M can pop the top symbol from the stack or push a word in Π^* on the top of the stack. It is well-known that counter machines with two counters have undecidable halting problem, and obviously the undecidability holds for machines augmented with a pushdown stack. Thus, we have to restrict the behaviors of the counters. A counter is *n-reversal-bounded* if it changes mode between nondecreasing and nonincreasing at most n times. For instance, the following sequence of counter values: $0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 3, 2, 1, 1, 1, 1, \cdots$ demonstrates only one counter reversal. A counter is *reversal-bounded* if it is *n-reversal-bounded* for some n that is independent of the computations. We note that a reversal-bounded M (i.e., each counter in M is reversal-bounded) does not necessarily limit the number of moves to be finite.

Note that the above defined M does not have an input tape; in this case it is used as a system specification rather than a language recognizer. When an NPCM M is used as a language recognizer we attach a separate one-way read-only input tape to the machine and assign a state in Q as the final (i.e., accepting) state. M *accepts* an input iff it can reach an accepting state. When M is reversal-bounded, the emptiness problem (i.e., whether M accepts some input) is known to be decidable,

Theorem 1. *The emptiness problem for reversal-bounded nondeterministic pushdown multicounter machines with a one-way input tape is decidable [21].*

An NCM can be regarded as an NPCM without the pushdown stack. Thus, the above theorem also holds for reversal-bounded NCMs. We can also give a nice characterization. If S is a set of n-tuples of integers, let $L(S)$ be the set of strings representing the tuples in S (each component, i.e., an integer, in a tuple is encoded as a unary string). It was shown in [21] that if $L(S)$ is accepted by a reversal-bounded nondeterministic multicounter machine, then S is semilinear. The converse is obvious. Since S is semilinear if and only if it is definable by a Presburger formula, we have:

Theorem 2. *A set of n-tuples of integers is definable by a Presburger formula iff it can be accepted by a reversal-bounded nondeterministic multicounter machine [21].*

3 Past Formulas

Let A be a finite set of finite state variables, i.e., their domains are a bounded range of integers. We use $a, b \cdots$ to denote them. Without loss of generality, we assume they are Boolean variables. All clocks are discrete. Let *now* be the clock representing the current time. Let X be a finite set of integer-valued variables. *Past-formulas* are defined as

$f ::= a(y) \mid y < n \mid y < z + n \mid \Box_x f[0, now] \mid f \vee f \mid \neg f,$

where $a \in A$, y and z are in $X \cup \{now\}$, $x \in X$, and n is an integer. Intuitively, $a(y)$ is the variable a's value at time y, i.e., $\texttt{Past}(a, y)$ in ASTRAL [8]. *Quantification* $\Box_x f[0, now]$, with $x \neq now$ (i.e., *now* can not be quantified), means, for all x from 0 to *now*, f holds. An appearance of x in $\Box_x f[0, now]$ is called *bounded*. We assume any x is bounded by at most one $\Box_x$. x is *free* in f if x is not bounded in f. f is *closed* if *now* is the only free variable. Past-formulas are interpreted on a history of Boolean variables. A history consists of a sequence of boolean values for each variable $a \in A$. The length of every sequence is $\mathbf{n}+1$, where $\mathbf{n}$ is the value of *now*. Formally, a *history* H is a pair $\langle \{\|a\|\}_{a \in A}, \mathbf{n} \rangle$, where $\mathbf{n} \in \mathbf{Z}^+$ is a nonnegative integer representing the value of *now*, and for each $a \in A$, the mapping $\|a\| : 0..\mathbf{n} \to \{0, 1\}$ gives the Boolean value of a at each time point from 0 to $\mathbf{n}$. Let $\mathcal{B} : X \to \mathbf{Z}^+$ be a *valuation* for variables in X. Thus $\mathcal{B}(x) \in \mathbf{Z}^+$ denotes the value of $x \in X$ under this valuation $\mathcal{B}$. We use $\mathcal{B}(n/x)$ to denote replacing x's value in the valuation with a non-negative integer n. Given a history H and a valuation $\mathcal{B}$, the interpretations of past-formulas are as follows, for each $y, z \in X \cup \{now\}$ and each $x \in X$,

$\|x\|_{H,\mathcal{B}} = \mathcal{B}(x)$, $\quad \|now\|_{H,\mathcal{B}} = \mathbf{n}$,

$\|a(y)\|_{H,\mathcal{B}} \iff \|a\|(\|y\|_{H,\mathcal{B}})$, $\quad \|y < n\|_{H,\mathcal{B}} \iff \|y\|_{H,\mathcal{B}} < n$,

$\|\neg f\|_{H,\mathcal{B}} \iff$ not $\|f\|_{H,\mathcal{B}}$, $\quad \|f_1 \vee f_2\|_{H,\mathcal{B}} \iff \|f_1\|_{H,\mathcal{B}}$ or $\|f_2\|_{H,\mathcal{B}}$.

$\|y < z + n\|_{H,\mathcal{B}} \iff \|y\|_{H,\mathcal{B}} < \|z\|_{H,\mathcal{B}} + n$,

$\|\Box_x f[0, now]\|_{H,\mathcal{B}} \iff$ for all k with $0 \leq k \leq \mathbf{n}$, $\|f\|_{H,\mathcal{B}(k/x)}$,

When f is a closed formula, we write $\|f\|_H$ instead of, for all $\mathcal{B}$, $\|f\|_{H,\mathcal{B}}$. We use $\Diamond_x$ to denote $\neg\Box_x\neg$.

A history H can be regarded as a sequence of snapshots $S_0, \cdots, S_{\mathbf{n}}$ such that each snapshot gives a value for each $a \in A$. When *now* progresses from $\mathbf{n}$ to $\mathbf{n}+1$ history H is updated to a new history H' by adding a new snapshot $S_{\mathbf{n}+1}$ to history H. This newly added snapshot represents the new values of $a \in A$ at the new current time $\mathbf{n}+1$. Is there any way to calculate the truth value of the formula under H' by using the new snapshot $S_{\mathbf{n}+1}$ and the truth value of the formula under H? If this can be done, the truth value of the formula can be updated along with the history's update from $\mathbf{n}$ to $\mathbf{n}+1$, without looking back at the old snapshots $S_0, \cdots, S_{\mathbf{n}}$. The rest of this section shows that this can be done.

A *Boolean function* is a mapping $\mathbf{Z}^+ \to \{0, 1\}$. A Boolean predicate is a mapping $\{0, 1\}^m \to \{0, 1\}$ for some m. We use $v_1, \cdots, v_{|A|}$ to denote the Boolean functions representing the truth value of each $a \in A$ at each time point. Obviously, $v_1, \cdots, v_{|A|}$ can be obtained by extending $\mathbf{n}$ to ∞ in a history. When $v_1, \cdots, v_{|A|}$ are given, a closed past formula can be regarded as a Boolean

function: the truth value of the formula at time t is the interpretation of the formula under the history $v_i(0), \cdots, v_i(t)$ for each $1 \leq i \leq |A|$. Given closed past formulas f, $g_1, \cdots, g_k$ (for some k), we use $u, u_1, \cdots, u_k$ to denote the Boolean functions for them, respectively.

Theorem 3. *For any closed past formula f, there are closed past formulas $g_1, \cdots, g_k$, and Boolean predicates $O, O_1, \cdots, O_k$ such that, for any given Boolean functions v_1, $\cdots$, $v_{|A|}$, the Boolean functions $u, u_1, \cdots, u_k$ (defined above) satisfy: for all t in $\mathbf{Z}^+$,*

$u(t+1) = O(v_1(t+1), \cdots, v_{|A|}(t+1), u_1(t), \cdots, u_k(t))$

and for each i, $1 \leq i \leq k$,

$u_i(t+1) = O_i(v_1(t+1), \cdots, v_{|A|}(t+1), u_1(t), \cdots, u_k(t))$.

Therefore, $u(t+1)$ (the truth value of formula f at time $t+1$), as well as each $u_i(t+1)$, can be recursively calculated by using the values of $v_1, \cdots, v_{|A|}$ at $t+1$, and values of $u_1, \cdots, u_k$ at t. As we mentioned before, past formulas can be expressed in TPTL [5]. A tableau technique is proposed in [5] to show that validity checking of TPTL is decidable. A modification of the technique can be used to prove Theorem 3.

To conclude this section, we point out that once the functions $v_1, \cdots, v_{|A|}$ representing each $a(now)$ for $a \in A$ are known, each closed past formula can be recursively calculated as in Theorem 3. In the next two sections, we will build past formulas into transition systems.

4 Past Machines: A Simpler Case

A *past-machine* M is a tuple $\langle S, A, E, now \rangle$, where S is a finite set of *(control) states*. A is a finite set of Boolean variables, and *now* is the only clock in M. *now* is used to indicate the current time. E is a finite set of *edges* or *transitions*. Each edge $\langle s, \lambda, l, s' \rangle$ denotes a transition from state s to state s' with *enabling condition* l and *assignments* λ to the Boolean variables in A. l is a closed past-formula. $\lambda : A \rightarrow \{0, 1\}$ denotes the new value $\lambda(a)$ of each variable $a \in A$ after an execution of the transition. Execution of a transition causes the clock *now* to progress by 1 time unit. A *configuration* α of M is a pair $\langle \alpha_{\mathbf{q}}, \alpha_H \rangle$ where $\alpha_{\mathbf{q}}$ is a state and $\alpha_H = \langle \{\|a\|^{\alpha_H}\}_{a \in A}, \mathbf{n}^{\alpha_H} \rangle$ is a history. $\alpha \rightarrow^{\langle s, \lambda, l, s' \rangle} \beta$ denotes a one-step transition along edge $\langle s, \lambda, l, s' \rangle$ in M satisfying:

- The state s is set to a new location, i.e., $\alpha_{\mathbf{q}} = s, \beta_{\mathbf{q}} = s'$.
- The enabling condition is satisfied, i.e., $\|l\|_{\alpha_H}$ holds under the history α_H.
- The clock *now* progresses by one time unit, i.e., $\mathbf{n}^{\beta_H} = \mathbf{n}^{\alpha_H} + 1$.
- The history α_H is extended to β_H by adding the resulting values (given by the assignment λ) of the Boolean variables after the transition. That is, for all $a \in A$, for all t, $0 \leq t \leq \mathbf{n}^{\alpha_H}$, history β_H is consistent with history α_H; i.e., $\|a\|^{\beta_H}(t) = \|a\|^{\alpha_H}(t)$. In addition, β_H extends α_H; i.e., for each $a \in A$, $\|a\|^{\beta_H}(\mathbf{n}^{\beta_H}) = \lambda(a)$.

We write $\alpha \rightarrow \beta$ if α can reach β by a one-step transition. A *path* $\alpha_0 \cdots \alpha_k$ satisfies $\alpha_i \rightarrow \alpha_{i+1}$ for each i. We write $\alpha \rightsquigarrow \beta$ if α reaches β through a path. α is *initial* if the current time is 0, i.e., $\mathbf{n}^{\alpha_H} = 0$. There are only finitely many initial configurations. Denote $R = \{\langle \alpha, \beta \rangle : \alpha \text{ is initial}, \alpha \rightsquigarrow \beta\}$.

Let M be a past machine as specified above. M, starting from an initial configuration α (i.e., with $now = 0$) can be simulated by a counter machine with reversal-bounded counters. In the following, we will show the construction. Each enabling condition l on an edge $e \in E$ of M is a closed past-formula. From Theorem 3, each l can be associated with a number of Boolean functions $O_l, O_{1,l}, \cdots, O_{k,l}$, and a number of Boolean variables $u_1^l, \cdots, u_k^l$ (updated while *now* progresses). l itself can be considered as a Boolean variable u^l. We use a primed form to indicate the previous value of a variable – here, a variable changes with time progressing. Thus, from Theorem 3,[1] these variables are updated as, $u^l := O_l(A, u_1^{l\prime}, \cdots, u_k^{l\prime})$ and for all u_i^l $u_i^l := O_{i,l}(A, u_1^{l\prime}, \cdots, u_k^{l\prime})$. Thus, M can be simulated by a counter machine M' as follows. M' is exactly the same as M except that each test of an enabling condition of l in M is replaced by a test of a Boolean variable u^l in M'. Furthermore, whenever M executes a transition, M' does the following (sequentially):

- increase the counter *now* by 1,
- change the values of Boolean variables $a \in A$ according to the assignment given in the transition in M,
- for each enabling condition of l in M, M' has Boolean variables $u^l, u_1^l, \cdots, u_k^l$. M' updates (as given above) $u_1^l, \cdots, u_k^l$ and u^l for **each** l. Of course, during the process, the new values of $a \in A$ will be used, which were already updated above.

The initial values of Boolean variables $a \in A$, u^l and u_j^l can be assigned using the initial values of $a \in A$ in α. M' contains only one counter *now*, which never reverses. Essentially M' is a finite state machine augmented by one reversal-bounded counter. It is obvious that M' faithfully simulates M. A configuration β can be encoded as a string composed of the control state β_q, the current time $\mathbf{n}^{\beta_H}$ (as a unary string), and the history concatenated by the values of $a \in A$ at time $0 \leq t \leq \mathbf{n}^{\beta_H}$. All components are separated by a delimiter "#" as follows:

$1^{\beta_q} \# \pi_0 \# \cdots \# \pi_{\mathbf{n}^{\beta_H}} \# 1^{\mathbf{n}^{\beta_H}}$

where π_t is a binary string with length $|A|$ indicating the values of all $a \in A$ at t. Thus, a set of configurations can be considered as a language. Denote R_α to be the set of configurations β with $\alpha \rightsquigarrow \beta$. Then,

Theorem 4. *R_α is accepted by a reversal-bounded nondeterministic multi-counter machine.*

Since there are only finitely many initial configurations, $R = \{\langle \alpha, \beta \rangle : \alpha \text{initial}, \beta \in R_\alpha\}$ can be accepted by a reversal-bounded nondeterministic multicounter machine.

[1] We simply use A to indicate the current values of $a \in A$ with the assumption that the "current time" can be figured out from the context.

Theorem 5. *R is accepted by a reversal-bounded nondeterministic multicounter machine.*

The reason that past machines are simple is that they contain only closed past formulas. Thus, we have to extend past machines by allowing a number of clock variables in the system, as shown in the next section.

5 Past Pushdown Timed Automata: A More Complex Case

Past machines can be extended by allowing extra free variables, in addition to *now*, in an enabling condition. We use $Z = \{z_1, \cdots, z_k\} \subseteq X$ to denote the variables other than *now*. A *past pushdown timed automaton* M is a tuple $\langle S, A, Z, \Pi, E, now\rangle$ where S and A are the same as those for a past machine. Z is a finite set of clocks with $now \notin Z$ and Π is a finite stack alphabet. Each edge, from state s to state s', in E is denoted by $\langle s, \delta, \lambda, (\eta, \eta'), l, s'\rangle$. l and λ have the same meaning as in a past machine, though the enabling condition l may contain, in addition to *now*, free (clock) variables in Z. $\delta \subseteq Z$ denotes a set of clock jumps. [2] δ may be empty. The stack operation is characterized by a pair (η, η') with $\eta \in \Pi$ and $\eta' \in \Pi^*$. That is, replacing the top symbol of the stack η by a word η'. A configuration α of M is a tuple $\langle \alpha_{\mathbf{q}}, \alpha_H, \alpha_Z, \alpha_w\rangle$ where $\alpha_{\mathbf{q}}$ is a state, α_H is a history as defined in a past machine, and $\alpha_Z \in (\mathbf{Z}^+)^{|Z|}$ is a valuation of clock variables Z. We use α_z to denote the value of $z \in Z$ under this configuration. $\alpha_w \in \Pi^*$ indicates the stack content.

$\alpha \rightarrow^{\langle s,\delta,\lambda,(\eta,\eta'),l,s'\rangle} \beta$ denotes a one-step transition along edge $\langle s, \delta, \lambda, (\eta, \eta'), l, s'\rangle$ in M satisfying:

- The state s is set to a new location, i.e., $\alpha_{\mathbf{q}} = s, \beta_{\mathbf{q}} = s'$.
- The enabling condition is satisfied, i.e., $\|l\|_{\alpha_H, \mathcal{B}(\alpha_Z/Z)}$ holds for any $\mathcal{B}$. That is, l is evaluated under the history α_H and replacing each free clock variable $z \in Z$ by the value α_z in the configuration α.
- Each clock changes according to the edge given.
  - If $\delta = \emptyset$, i.e., there are no clock jumps on the edge, then the now-clock progresses by one time unit. That is, $\mathbf{n}^{\beta_H} = \mathbf{n}^{\alpha_H} + 1$. All the other clocks do not change; i.e., for each $z \in Z$, $\beta_z = \alpha_z$.
  - If $\delta \neq \emptyset$, then all the clocks in δ jump to *now*, and the other clocks do not change. That is, for each $z \in \delta$, $\beta_z = \mathbf{n}^{\alpha_H}$. In addition, for each $z \notin \delta$, $\beta_z = \alpha_z$, and the clock *now* does not progress, i.e., $\mathbf{n}^{\beta_H} = \mathbf{n}^{\alpha_H}$.
- The history is updated similarly as for past machines. That is,
  - If $\delta = \emptyset$, then *now* progresses, for all $a \in A$, for all t, $0 \leq t \leq \mathbf{n}^{\alpha_H}$, $\|a\|^{\beta_H}(t) = \|a\|^{\alpha_H}(t)$, and $\|a\|^{\beta_H}(\mathbf{n}^{\beta_H}) = \lambda(a)$.

[2] Here we use clock jumps (i.e., $x := now$) instead of clock resets ($x := 0$). The reason is that, in this way, the start time of a transition can be directly modeled as a clock. Obviously, a transform from x to $now - x$ will give a "traditional" clock with resets.

- If $\delta \neq \emptyset$, then *now* does not progress, for all $a \in A$, for all t, $0 \leq t \leq \mathbf{n}^{\alpha_H} - 1$, $\|a\|^{\beta_H}(t) = \|a\|^{\alpha_H}(t)$, and $\|a\|^{\beta_H}(\mathbf{n}^{\beta_H}) = \lambda(a)$. Thus, even though the now-clock does not progress, the current values of variables $a \in A$ may change according to the assignment λ.

– According to the stack operation (η, η'), the stack word α_w is updated to β_w.

α is initial if the stack word is empty and all clocks including *now* are 0. Similar to the case for past machines, we define $\alpha \rightsquigarrow \beta$ and R. Again, R can be considered as a language by encoding a configuration into a string. The main result in this section is that R can be accepted by a reversal-bounded NPCM. The major difference between a past machine and a past pushdown timed automaton is that the enabling condition on an edge in the past pushdown timed automaton is not necessarily a closed past formula. The proof, which can be found in [11], shows that an enabling condition l with free variables in Z can be made closed.

Theorem 6. *The set R of a past pushdown timed automaton can be accepted by a reversal-bounded NPCM.*

The importance of the automata-theoretic characterization of R is that the Presburger safety properties over clocks and stack word counts are decidable. We use β to denote variables ranging over configurations. We use $\mathbf{q}, \mathbf{z}, \mathbf{w}$ to denote variables ranging over control states, clock values and stack words, respectively. Note that β_{z_i}, β_q and β_w are still used to denote the value of clock z_i, the control state and the stack word of β. We use a count variable $\#_a(\mathbf{w})$ to denote the number of occurrences of a character $a \in \Pi$ in a stack word variable $\mathbf{w}$. An NPCM-term t is defined as follows: [3] $t ::= n \mid \mathbf{q} \mid \mathbf{x} \mid \#_a(\beta_w) \mid \beta_{x_i} \mid \beta_q \mid t{-}t \mid t{+}t$, where n is an integer and $a \in \Pi$. An NPCM-formula P is defined as follows: $P ::= t > 0 \mid t \bmod n = 0 \mid \neg P \mid P \vee P$, where $n \neq 0$ is an integer. Thus, P is a Presburger formula over control state variables, clock value variables and count variables. The Presburger safety analysis problem is: given a past pushdown timed automata and an NPCM-formula P, is there a reachable configuration satisfying P? From Theorem 1, Theorem 2, Theorem 6, and the proof of the Theorem 10 in [15], we have,

Theorem 7. *The Presburger safety analysis problem for past pushdown timed automata is decidable.*

Because of Theorem 7, the following statement can be verified:

"A given past pushdown timed automaton can reach a configuration satisfying $z_1 - z_2 + 2z_3 > \#_a(w) - 4\#_b(w)$."

where z_1, z_2, and z_3 are clocks and w is the stack word in the configuration. Obviously, Theorem 6 and Theorem 7 still hold when a past pushdown timed automaton is augmented with a number of reversal-bounded counters, i.e., a past reversal-bounded pushdown timed automaton. The reason is as follows. In the

[3] Control states can be interpreted over a bounded range of integers. Therefore, an arithmetic operation on control states is well-defined.

proof of Theorem 6, clocks in a past pushdown timed automaton are simulated by reversal-bounded counters. Therefore, by replacing past-formulas in a past pushdown timed automaton with Boolean formulas in the proof, a past pushdown timed automaton is simulated by a reversal-bounded NPCM. When a number of reversal-bounded counters are added to the past pushdown timed automaton, the automaton can still be simulated by a reversal-bounded NPCM: clocks are simulated by reversal-bounded counters and the added reversal-bounded counters remain. Hence, Theorem 6 and Theorem 7 still hold for past reversal-bounded pushdown timed automata. An unrestricted counter is a special case of a pushdown stack. Therefore, the results for past reversal-bounded pushdown timed automata imply the same results for past timed automata with a number of reversal-bounded counters and an unrestricted counter. These results are helpful in verifying Presburger safety properties for history-dependent systems containing parameterized (unspecified) integer constants, as illustrated by the example in the next section.

6 An Example

This section considers an ASTRAL specification [22] of a railroad crossing system, which is a history-dependent and parameterized real-time system with a Presburger safety property that needs to be verified. The system description is taken from [17]. The system consists of a set of railroad tracks that intersect a street where cars may cross the tracks. A gate is located at the crossing to prevent cars from crossing the tracks when a train is near. A sensor on each track detects the arrival of trains on that track. The critical requirement of the system is that whenever a train is in the crossing the gate must be down, and when no train has been in between the sensors and the crossing for a reasonable amount of time, the gate must be up. The complete ASTRAL specification of the railroad crossing system can be found in [22] and at `http://www.cs.ucsb.edu/~dang`. The ASTRAL specification was proved to be correct by using the PVS-based ASTRAL theorem prover [22] and was tested by a bounded-depth symbolic search technique [14].

The ASTRAL specification looks at the railroad crossing system as two interactive modules or process specifications: `Gate` and `Sensor`. Each process has its own (parameterized) constants, local variables and transition system. Requirement descriptions are also included as a part of a process specification. ASTRAL is a rich language and has strong expressive power. For a detailed introduction to ASTRAL and its formal semantics the reader is referred to [8, 10, 22]. For the purpose of this paper, we will show that the `Gate` process can be modeled as a past pushdown timed automaton with reversal-bounded counters. By using the results in the previous section, a Presburger safety property specified in `Gate` can be automatically verified.

We look at an instance of the `Gate` process by considering the specification with one railroad track (i.e., `n_track`=1, and therefore there is only one `Sensor` process instance.) and assigning concrete values to parameterized constants as

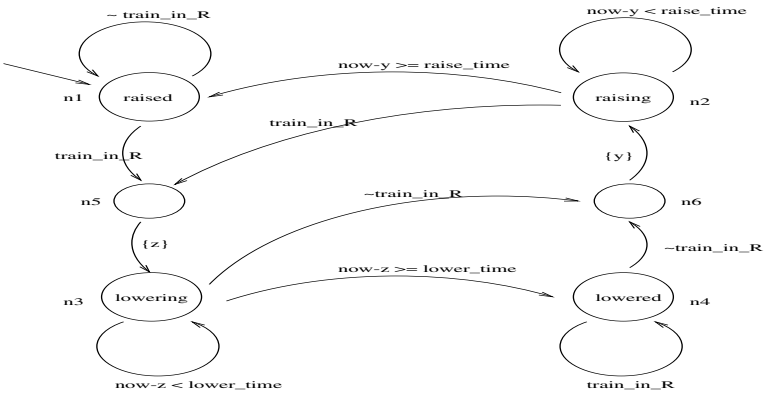


Fig. 1. The transition system of a `Gate` instance represented as a timed automaton

follows (in order to make the enabling conditions in the process in the form of past formulas):

`raise_dur`=1, `up_dur`=1, `lower_dur`=1, `down_dur`=1,
`raise_time`=1, `lower_time`=1, `response_time`=1, `RIImax`=6.

But two constants `wait_time` and `RImax` remain parameterized.

The transition system of the `Gate` process can be represented as the timed automaton shown in Figure 1. The local variable `position` in `Gate` has four possible values. They are `raised`, `raising`, `lowering` and `lowered`, which are represented by nodes n_1, n_2, n_3 and n_4 in the figure, respectively. There are two dummy nodes n_5 and n_6 in the graph, which will be made clear in a moment. The initial node is n_1. That is, the initial position of the gate is `raised`.

The transitions `lower, down, raise` and `up` in the `Gate` process are represented in the figure as follows. Each transition includes a pair of entry and exit assertions with a nonzero duration associated with each pair. The entry assertion must be satisfied at the time the transition starts, whereas the exit assertion will hold after the time indicated by the duration from when the transition fires. The transition `lower`,

```
TRANSITION lower
ENTRY [ TIME : lower_dur ]
  ~ ( position = lowering | position = lowered )
  & EXISTS s: sensor_id ( s.train_in_R )
EXIT
  position = lowering,
```

corresponds to the edges $\langle n_1, n_5\rangle$ and $\langle n_5, n_3\rangle$, or the edges $\langle n_2, n_5\rangle$ and $\langle n_5, n_3\rangle$. The clock z is used to indicate the end time `End(lower)` (of transition `lower`) used in transition `down`. Whenever the transition `lower` completes, z jumps to *now*. Thus, a dummy node n_5 is introduced such that z jumps on the edge $\langle n_5, n_3\rangle$ to indicate the end of the transition `lower`. On an edge without clock

jumps (such as $\langle n_1, n_5\rangle$ and $\langle n_2, n_5\rangle$), *now* progresses by one time unit. Thus, the two edges $\langle n_1, n_5\rangle$ and $\langle n_2, n_5\rangle$ indicate the duration `lower_dur` of the transition `lower` (recall the parameterized constant `lower_dur` was set to be 1.).

Similarly, transition `raise` corresponds to the edges $\langle n_3, n_6\rangle$ and $\langle n_6, n_2\rangle$, or the edges $\langle n_4, n_6\rangle$ and $\langle n_6, n_2\rangle$. The other two transitions `down` and `up` correspond to the edges $\langle n_3, n_4\rangle$ and $\langle n_2, n_1\rangle$, respectively. Idle transitions need to be added to indicate the behavior of the process when no transition is enabled and executing. They are represented by self-loops on nodes n_1, n_2, n_3 and n_4 in the figure.

Besides variable `position`, `Gate` has an imported variable `train_in_R`, which is a local variable of the `Sensor` process, to indicate an arrival of a train. `Gate` has no control over the imported variable. That is, `train_in_R` can be either true or false at any given time, even though we do not explicitly specify this in the figure. But not all the execution sequences of the `Gate` process are intended. For instance, consider the scenario that `train_in_R` has value *true* at $now = 2$ and the value changes to $false$ at $now = 3$. This change is too fast, since the gate `position` at $now = 3$ may be `lowering` when the change happens. At $now = 3$, the train had already crossed the intersection. This is bad, since the gate was not in the fully lowered position `lowered`. Thus, the imported variable clause is needed to place extra requirements on the behaviors of the imported variable. The requirement essentially states that once the sensor reports a train's arrival, it will keep reporting a train at least as long as it takes the fastest train to exit the region. By substituting for the parameterized constants and noticing that there is only one sensor in the system, the imported variable clause in the ASTRAL specification can be written as

$$now \geq 1 \wedge \ \texttt{past}(\texttt{train_in_R}, now - 1) = true \wedge \ \texttt{train_in_R} = false$$
$$\rightarrow$$
$$now \geq 5 \wedge \ \forall t(t \geq now - 5 \wedge t < now \ \rightarrow \ \texttt{past}(\texttt{train_in_R}, t) = true).$$

We use f to denote this clause. It is easy to see that f is a past formula. Figure 1 can be modified by adding f to the enabling condition of each edge. The resulting automaton is denoted by M.

It is easy to check that M does rule out the unwanted execution sequences shown above. Now we use clock x to indicate the (last) change time of the imported variable `train_in_R`. A proper modification to M can be made by incorporating clock x into the automaton. The resulting automaton, denoted by M', is a past pushdown timed automaton without the pushdown stack. Recall that the process instance has two parameterized constants `wait_time` and `RImax`. Therefore, M' is augmented with two reversal-bounded counters `wait_time` and `RImax` to indicate the two constants. These two counters remain unchanged during the computations of M' (i.e., 0-reversal-bounded). They are restricted by the axiom clause g of the process:

```
   wait_time >= raise_dur + raise_time + up_dur
& RImax >= response_time + lower_dur +
      lower_time + down_dur + raise_dur
& RImax >= response_time + lower_dur +
      lower_time + down_dur + up_dur
```

recalling that all the constants in the clause have concrete values except `wait_time` and `RImax`.

The first conjunction of the schedule clause of the process instance specifies a safety property such that the gate will be down before the fastest train reaches the crossing; i.e., $(\texttt{train_in_R} = \mathit{true} \ \wedge \ \mathit{now} - x \geq \texttt{RImax} - 1) \rightarrow \texttt{position} = \texttt{lowered}$. We use p to denote this formula. Notice that p is a non-region property (since `RImax` is a parameterized constant). Verifying this part of the schedule clause is equivalent to solving the Presburger safety analysis problem for M' (augmented with two reversal-bounded counters) with the Presburger safety property $g \rightarrow p$ over the clocks and the reversal-bounded counters. From the result of the previous section, this property can be automatically verified.

Acknowledgement

The authors would like to thank P. San Pietro and J. Su for discussions. The ASTRAL specification used in this paper was written by P. Kolano. The work by Dang and Kemmerer was supported by DARPA F30602-97-1-0207. The work by Bultan was supported by NSF CCR-9970976 and NSF CAREER award CCR-9984822. The work by Ibarra was supported by NSF IRI-9700370.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, *"Model-checking in dense real time,"* *Information and Computation*, **104** (1993) 2-34 74

[2] R. Alur and D. Dill, *"A theory of timed automata,"* *TCS*, **126** (1994) 183-236 74

[3] R. Alur, T. Feder, and T. A. Henzinger, *"The benefits of relaxing punctuality,"* *J. ACM*, **43** (1996) 116-146 74

[4] R. Alur, T. A. Henzinger, *"Real-time logics: complexity and expressiveness,"* *Information and Computation*, **104** (1993) 35-77 74

[5] R. Alur, T. A. Henzinger, *"A really temporal logic,"* *J. ACM*, **41** (1994) 181-204 74, 75, 78

[6] A. Bouajjani, J. Esparza, and O. Maler, *"Reachability analysis of pushdown automata: application to model-checking,"*, *CONCUR'97*, LNCS **1243**, pp. 135-150 75

[7] A. Bouajjani, R. Echahed, and R. Robbana, *"On the automatic verification of systems with continuous variables and unbounded discrete data structures,"* *Hybrid System II*, LNCS **999**, 1995, pp. 64-85 75

[8] A. Coen-Porisini, C. Ghezzi and R. Kemmerer, *"Specification of real-time systems using ASTRAL,"* *TSE*, **23** (1997) 572-598 75, 77, 82

[9] H. Comon and Y. Jurski, *"Timed automata and the theory of real numbers,"* *CONCUR'99*, LNCS **1664**, pp. 242-257 74

[10] A. Coen-Porisini, R. Kemmerer and D. Mandrioli, *"A formal framework for ASTRAL intralevel proof obligations,"* *TSE*, **20** (1994) 548-561 75, 82

[11] Z. Dang, *"Verification and debugging of infinite state real-time systems,"* PhD Dissertation, UCSB, August 2000. Available at `http://www.cs.ucsb.edu/~dang` 75, 81

[12] Z. Dang, *"Binary reachability analysis of timed pushdown automata with dense clocks," CAV'01*, LNCS **2102**, pp. 506-517 74
[13] Z. Dang and R. A. Kemmerer, *"Using the ASTRAL model checker to analyze Mobile IP," ICSE'99*, pp. 132-141 75
[14] Z. Dang and R. A. Kemmerer, *"Using the ASTRAL symbolic model checker as a specification debugger: three approximation techniques," ICSE'00*, pp. 345-354 82
[15] Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer and J. Su, *"Binary reachability analysis of discrete pushdown timed automata," CAV'00*, LNCS **1855**, pp. 69-84 74, 81
[16] A. Finkel, B. Willems and P. Wolper, *"A direct symbolic approach to model checking pushdown systems,"* INFINITY'97 75
[17] C. Heitmeyer and N. Lynch. "The generalized railroad crossing: a case study in formal verification of real-time systems," *RTSS'94*, pp. 120-131 82
[18] T. A. Henzinger, Z. Manna, and A. Pnueli, *"What good are digital clocks?," ICALP'92*, LNCS **623**, pp. 545-558
[19] T. A. Henzinger and Pei-Hsin Ho, *"HyTech: the Cornell hybrid technology tool," Hybrid Systems II*, LNCS **999**, 1995, pp. 265-294 74
[20] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, *"Symbolic model checking for real-time systems," Information and Computation*, **111** (1994) 193-244 74
[21] O. H. Ibarra, *"Reversal-bounded multicounter machines and their decision problems," J. ACM*, **25** (1978) 116-133 75, 76
[22] P. Z. Kolano, Z. Dang and R. A. Kemmerer. "The design and analysis of real-time systems using the ASTRAL software development environment," *Annals of Software Engineering*, **7** (1999) 177-210 82
[23] K. G. Larsen, P. Pattersson, and W. Yi, *"UPPAAL in a nutshell," International Journal on Software Tools for Technology Transfer*, **1** (1997) 134-152 74
[24] F. Laroussinie, K. G. Larsen, and C. Weise, *"From timed automata to logic - and back," MFCS'95*, LNCS **969**, pp. 529-539 74
[25] Amir Pnueli, *"The temporal logic of programs," FOCS'77*, pp. 46–57 75
[26] J. Raskin and P. Schobben, *"State clock logic: a decidable real-time logic," HART'97*, LNCS 1201, pp. 33-47 74
[27] H. Straubing, *Finite automata, formal logic, and circuit complexity.* Birkhauser, 1994 75
[28] W. Thomas, *"Automata on infinite objects,"* in *Handbook of Theoretical Computer Science, Volume B* (J. van Leeuwen eds.), Elsevier, 1990 75
[29] T. Wilke, *"Specifying timed state sequences in powerful decidable logics and timed automata,"* LNCS 863, pp. 694-715, 1994 74
[30] S. Yovine, *"A verification tool for real-time systems," International Journal on Software Tools for Technology Transfer*, **1** (1997): 123-133 74

Scheduling Hard Sporadic Tasks by Means of Finite Automata and Generating Functions

Jean-Philippe Dubernard[1] and Dominique Geniet[2]

[1] L.I.F.A.R., Université de Rouen
Place Émile Blondel, F-76821 Mont SaintAignan Cédex
jean-philippe.dubernard@univ-rouen.fr
[2] L.I.S.I., Université de Poitiers & E.N.S.M.A.
Téléport 2, 1 av. Clément Ader, BP 40109, F-86961 Futuroscope Chasseneuil Cédex
dgeniet@ensma.fr

Abstract. In a previous work, we propose a technique to decide feasability of periodic hard real-time systems based on finite automata. Here, associating generating functions (whose role is "to predict the future") to a finite automaton, we extend this technique to hard sporadic tasks, independent or interdependent with the periodic tasks.

1 Introduction

A control-command system is a **real-time** system if its speed is constrained by the movements of the drived physical process in its universe. Such a system is **reactive** (because of the input signals, which must be processed *instantaneously*) and **concurrent** (because the different decisional computings concerning the physical process driving must be processes *simultaneously*). Then, a real-time system is composed of basic **tasks**. Each of these tasks implements the computing of a reaction (or a part of a reaction) to an **event** of the form *data entrance* or *set of datas entrances*.

Some informations regularly come in the system (mainly data coming from captors). So, the tasks in charge of their capture and their treatment are naturally periodically activated, and the period is determinated by the captor frequency.

Other information comes in the system in an unpredictable way: alarm signals, for example, but also the human supervisor. The arrival of such events generates the activation of non periodic tasks that are usually classed into two groups [SSRB98]:

– **aperiodic** tasks for which the sole known information is its computation time[1] C. Generally, these tasks are not considered to be hard tasks, and they do not interact with the periodic part of the application.

[1] Allocation period of the CPU necessary to the complete execution of an occurence of the task.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 87–100, 2002.
© Springer-Verlag Berlin Heidelberg 2002

– **sporadic** tasks, for which we get the allocation time C, the relative deadline[2] D and the pseudo-period[3] P.

The *temporal validation* of a real-time system consists in proving that, whatever the level of the input event flow, the real-time system will always react according to its temporal specifications.

Of course, this step of the development of a real-time system is crucial. The problem of the scheduling[4] of a task configuration is, as a general rule, NP-hard [Mok83, LM80]. Many works of the real-time community stand on the characterization of the scheduling by analytic (and simulation) techniques (one can find a survey on on-line and off-line techniques in [Gro99] for example). For periodic systems -i.e., only composed of periodic tasks- we dispose of a set of on-line scheduling algorithms (see [But97], [Bak91] and [ABD+95]) which adresses the set of problems connected to real situations (interdependency, deadlocks avoidance, importance of the tasks, etc.). [Gro99] gives an algorithm which computes the minimal simulation duration to valid interdependent periodic systems, in the centralized case.

As far as we know, there are very few works which adress the problem of the case of the interdependent sporadic tasks [SCE90]. Here, we deal with this problem. So, in the following, we consider that a real-time system is composed of two sets of tasks: $\{(\tau_i)_{i\in[1,n]}\}$ and $\{(\alpha_i)_{i\in[1,p]}\}$. The τ_i's are periodic and the α_i's sporadic task. All these tasks are sequential: they cannot be not parallelized. They can communicate and share resources. Each task is temporally characterized by a 4-tuple (r_i, C_i, D_i, P_i) [LL73], whose semantics is given in Fig. 1. The differences between the τ_i's and the α_i's are the following:

– for the τ_i's, the time r_i of first activation is statically known: its numerical value is acquired before the application starts. If we denote by $\delta_{i,j}$ the activation time of the j-th instance of τ_i, we get $\delta_{i,j+1} = \delta_{i,j} + P_i$. For the α_i's, this time is unknown: if not we could statically predict the future behaviour of the process and then, the occurence time of alarm signals.
– for the τ_i's, the period P_i is fixed: it is an integer fixed statically. If we denote by $\delta_{i,j}$ the activation time of the j-th instance of α_i, we obtain $\delta_{i,j+1} \geq \delta_{i,j} + P_i$. For the α_i's, this integer indicates the minimal delay which separates two successive activation dates.

Moreover, we suppose that tasks are not reentrant. Then, we suppose that $P_i \geq D_i$, for all i in $[1, n + p]$. At last, we consider multi-processor target architectures with shared memory. We make the hypothesis of the total migration: an instance of a task can migrate during its execution.

Here, we use the temporal model of real-time systems based on regular languages, introduced in [Gen00]. In this model, time is implicit. This approach

[2] Duration which separates the activation time of an occurence of the task and its absolute deadline.

[3] Minimal delay separating two successive activations of the task.

[4] A system is feasable if there exists at least one way to schedule its tasks according to its temporal constraints.

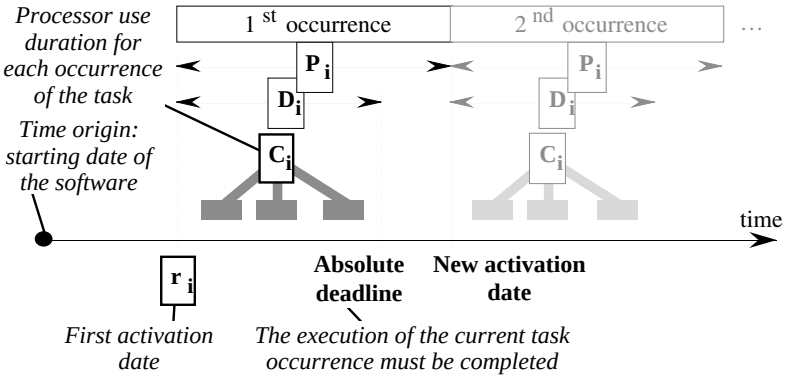


Fig. 1. Temporal model of tasks

avoids some problems usually set by other model oriented approaches (temporized automata [ABBL98], Petri nets [GCG00], for instance). On the one hand, it gives decision processes based on the finite automata analysis techniques, whose power, efficacy and modularity have been established a long time ago. For example, the main result of [Gro99] is the cyclicity of the scheduling sequences in monoprocessor. In our approach, it comes from the property *the set of the valid behaviours of a real-time application is a regular language*. As far as this property holds for multiprocessor target architectures, we can say that the cyclicity of the scheduling also remains valid in multiprocessor.

On the other hand, quite thin properties can be modeled in our approach, which are more difficult to reach by other methods because they need either some elaborated equational handlings (temporized approaches) or some quite heavy structural definitions (Petri nets feasability decision approaches).

In the first section, from an example, we introduce the regular language based model and the algorithm. Next we show how to integrate sporadic tasks in our model and we extend the method of decision to the systems which contain both sporadic task and periodic tasks. Finally, we evaluate, from a complexity study, how to use this methodology on realistic cases.

2 A Periodic Real-Time System

The temporal model using regular languages is based on the definition of **valid temporal behaviour**. Let us consider a task τ_i, which starts its execution at time t_0 (we consider a discrete space time) and let $t > t_0$ be the time when τ_i is observed. The sequence ω_t of the statements executed by τ_i between t_0 and t is a prefix of the behaviour of τ_i. The word ω_t is a **valid behaviour** of τ_i if there exists at least one way to extend ω_t on an arbitrary long derivation such that τ_i respects its temporal constraints. So, a valid behaviour is a word of P^*, where P is the set of the basic statements of the target machine. We make a partition

of $P = P_c \cup P_n$. P_c is the set of critical statements: P/V for a resource, S/R [5] for a message. P_n is the set of all other statements. Now, let us introduce two symbols, a_i and $\bullet$, and let us consider the morphism $\mu : P \to P_c \cup \{a_i\}$ such that $x \in P_c \Leftrightarrow \mu(x) = x$ and $x \notin P_c \Leftrightarrow \mu(x) = a_i$. We call **valid temporal behaviour** of τ_i the shuffle set $\mu(\omega_t) \text{Ш} \bullet^{t-|\omega_t|}$: a_i symbolizes the activity of τ_i for one time unit, and $\bullet$ its inactivity (the task is suspended) for one time unit. Let us denote by Σ the set $P_c \cup \{a_i, \bullet\}$. We show, in [Gen00], that the set of valid temporal behaviours of τ_i is the set of the words of the center[6] language $Center\left(\bullet^{r_i}.\left(\left(\omega_{P_i} \text{Ш} \bullet^{D_i - C_i}\right).\bullet^{P_i - D_i}\right)^*\right)$. This language can be refined to model specific properties of τ_i's behaviour (for example, the non-preemptibility when in critical section). Finally, we build a finite automaton which accepts this language.

2.1 An Example

Let us consider a system of periodic tasks $\{\tau_1, \tau_2\}$, where $(r_1, C_1, D_1, P_1) = (1, 3, 3, 6)$ and $(r_2, C_2, D_2, P_2) = (2, 3, 6, 8)$. τ_1 uses, in a non preemptive way, the resource R_1 during its two first running time units. τ_2 also uses R_1, in a non preemptive way, during its two last running time units.

For τ_1, we get $\omega_{P_1} = p_1.v_1.a_1$. For τ_2, we get $\omega_{P_2} = a_2.p_1.v_1$. The associated languages are accepted by the finite automata[7] A_1 and A_2 given in Fig. 2. All the states of these automata are terminal.

Let us note L_1 (resp. L_2) the language associated with A_1 (resp. A_2). The behaviour of the ressource R_1 is represented by the automaton given in Fig. 2: R_1 can be *free* (initial state) or *busy* (non initial state). From the initial state, R_1 can *stay free* (action f_1) or *become busy* (action p_1). We get a similar behaviour for the non initial state, with *to stay busy* or *to free itself*.

Our approach differs from the usual modeling of programs by finite (temporized or not) automata, in the sense that our automaton is not a functional model of the program executed by τ_i. We are only interested in the state of the task (as an object handled by the scheduler): **active** or **inactive**. So, we leave the functional semantics of the program. This comes from the fact that our objective is neither the study of functional (or behaviour) properties, nor an on line driving of the tasks system. We are interested in defining the class of temporal properties that can be decided from an algebraic structure as simple as possible.

[5] P is the **Semaphore Get** statement, and V the **Semaphore Release** statement. These statements were defined [Dij65] to guarantee the mutual exclusion on the access to the resource controled by the semaphore concerned. For message communications, tasks use the statements S (**Send** statement) and R (**receive** statement).

[6] If L is a language, $Center(L)$ is the set $\{\xi \in \Sigma^*, \forall n \in \mathbb{N}\ \exists \chi, \xi.\chi \in L \text{ and } |\xi.\chi| > n\}$: these are the prefix words of L which can be prolonged as far as we want. Note that $|\omega|$ is the length of the word ω and $|\omega|_a$ is the number of letters a in ω.

[7] Concerning finite automata, we use the definition given in [Eil76].

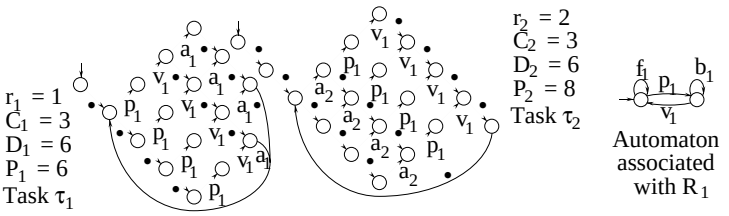


Fig. 2. Automata associated with periodic tasks

2.2 Feasability of the System $\{\tau_1, \tau_2\}$

To represent the behaviours of $\{\tau_1, \tau_2\}$, we build, first, the automaton A_T of the system $\{\tau_1, \tau_2, R_1\}$. The labels of A_T's transitions are of the form (x, y, z), where x (resp. y) is the status of τ_1 (resp. τ_2) and z the status of R_1. This technique comes from the Arnold-Nivat model [Arn94], defined for the synchronization of processes: we build a sub-graph A_S of A_T, restricted to the transitions whose label satisfies:

$$\forall a \ \in \{p_i, v_i\}, z = a \Leftrightarrow (x = a \ xor \ y = a)$$

Thus, we consider the sub-part of A_T which satisfies:

- at most one of the tasks runs a critical statement;
- if a task runs a critical statement, the resource model task runs it too.

This operation consists (algebraically) in making an intersection of languages. This operation is not internal in the class of centers of regular languages: thus, we build the automaton which accepts $Center(L(A_S))$. Each word of this language corresponds to a sequence of scheduling. Moreover, we know, by construction of the language, that this sequence can be extended in order to lead each task to respect its temporal constraints. So, the schedulability criterion of the task system is $L(A_S) \neq \emptyset$. On automata, this criterion is implemented by the test $\{final \ states\} \neq \emptyset$. The third component (the resource model task) is now useless: it can be deleted, applying to the set of the transition labels, the projection: $\pi_{1,2} : (x, y, z) \rightarrow (x, y)$. As this operation is a morphism, the image language is a center.

The hardware architecture is also considered. The automaton which accepts the language $\pi_{1,2}(Center(L(A_S)))$ gets transitions labelled by couples (x, y): x (resp. y) is the status of τ_1 (resp. τ_2). To obtain the automaton accepting the schedulability sequences in mono-processor (resp. as a general rule, on k processors), we delete the set of transitions whose label does not get any $\bullet$ (resp. the set of transitions whose label $(x_i)_{i\in[1,n]}$ gets less than $n-k$ $\bullet$). As this operation is also implemented through an intersection of languages, it must be followed by the computation of a center of language. Applied to the example $\{\tau_1, \tau_2\}$, this technique shows that the tasks can be scheduled in mono-processor: we obtain

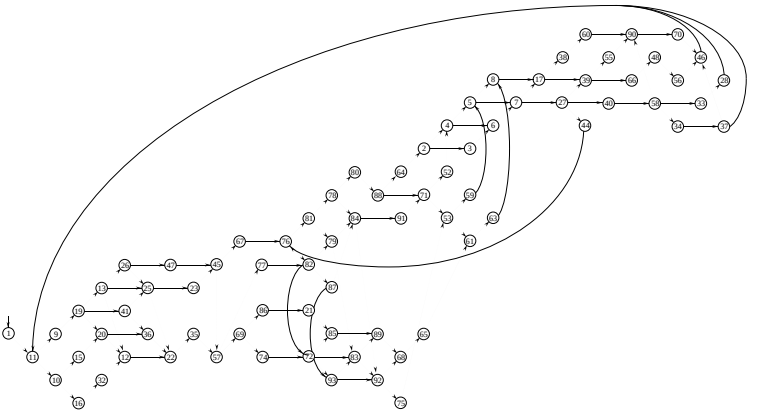


Fig. 3. Automaton of the system $\{\tau_1, \tau_2\}$ in mono-processor

an automaton whose topology is given in Fig. 3, and whose accepated language is not empty.

In the following, we denote by $A_k(S)$ the automaton of acceptance of the schedulability sequences of the system S on an architecture with k processors. Thus, the automaton given in Fig. 3 is $A_1(\{\tau_1, \tau_2\})$

3 Sporadic Tasks

The sporadic tasks differ from the periodic ones by two characteristics:

- a priori, one does not know their first activation date;
- for the periodic tasks, P_i is the fixed duration between two succesive activations; for the sporadic tasks, it is a minimal delay between two successive activation dates of the task.

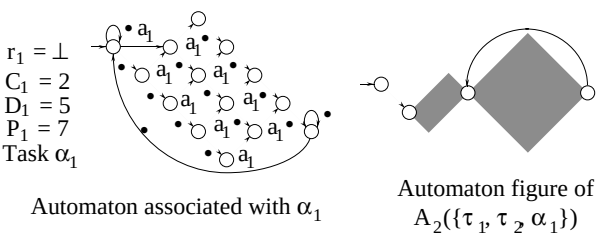


Fig. 4. An example

3.1 Automata Associated with Sporadic Tasks

For periodic tasks, we have seen above that represents the regular expression of the regular language which collects the valid behaviours is $Center\left(\bullet^{r_i}.\left(\left(\omega_{P_i} \text{ш} \bullet^{D_i-C_i}\right).\bullet^{P_i-D_i}\right)^*\right)$. For a sporadic task, r_i is not known. The prefix period can be any integer. Thus, the prefix is $\bullet^*$. The sporadic tasks that we consider here are hard tasks. So, they are characterised by a word ω_{P_i} of length C_i and by a deadline. Thus, the activation sequence of such a task is the same as for a periodic task. The duration of inactivity of a sporadic task between the deadline of an instance and the activation of the following one is an integer greater (or equal) than P_i. The corresponding word gets the form $\bullet^n$, where $n \geq P_i$. As n is not statically known, the suffix is $\bullet^n.\bullet^*$. The proof of [Gen00] concerning the structure of the language of valid behaviours is also available in this case. So, the language of valid temporal behaviours of a sporadic task α_i, with temporal characteristics (C_i, D_i, P_i) is $Center\left(\bullet^*.\left(\left(\omega_{P_i} \text{ш} \bullet^{D_i-C_i}\right).\bullet^{P_i}\right)^*\right)$. As this expression is regular, it naturally provides an automaton of acceptance for this language. For example, for the task α_1 with temporal characteristics $(C_1, D_1, P_1) = (2, 5, 7)$, this automaton is given on Fig. 4 left.

3.2 Schedulability for a System which Integrates Sporadic Tasks

In the general case, the methodology introduced in [Gen00] integrates any type of task whose temporal properties allow a representation by finite automata. Then, the mechanism of calculus of $A_k(S)$ can be used in the case where S integrates sporadic task and periodic tasks, possibly with interdependence between these different tasks. Let us denote by $\diamond$ the operation *synchronized product* which, from $A_k(S_1)$ and $A_k(S_2)$, computes $A_k(S_1 \cup S_2)$. Any system S is composed of a periodic sub-system S_P and of a sub-system S_S which gets sporadic tasks. The commutativity of algebraic operations used here naturally leads to $A_k(S) = A_k(S_P) \diamond A_k(S_S)$. For example, if we consider the system τ_1, τ_2, α_1, the automaton $A_1(S_P)$ gets 93 states and 136 transitions; it is given in Fig. 3. $A_1(S_S)$ is the automaton constituted by 1674 states and 3867 transitions; its figure is given in Fig. 4 right. This system of tasks can also be scheduled on a monoprocessor: $A_1(S)$ is an automaton containing 1277 states and 1900 transitions.

3.3 Schedulability Desision

General Case The method presented in Section 3.2 is the general case where the τ_i's and the α_j's are interdependent. Thus, the schedulability criterion of a system valid for tasks is the same than in the case of periodic systems: *S is schedulable on k processors* $\Leftrightarrow L(A_k(S)) \neq \emptyset$.

However, in real cases, the τ_i's and the α_j's are frequently independent: this particular case motivates a special study. In this case, the synchronized product $A_k(S_P) \diamond A_k(S_S)$ does not bring any information about the interdependence of tasks, but only a decision of schedulability about processor sharing.

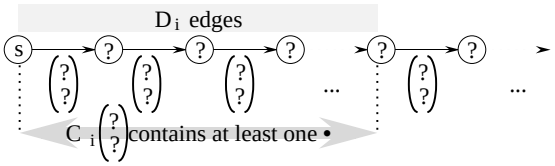


Fig. 5. Schedulability expression on the languages

In this precise case, the schedulability can be decided, for a set of sporadic tasks, by giving only:

- the automaton $A_k(S_P)$,
- the set of the (C_j, D_j, P_j) characterizing the sporadic tasks α_j,
- an on-line approach for the use of $A_k(S_P)$.

Improvement of the Complexity when the Sporadic Tasks Are Interdependent Here, no α_j is interdependent with another task (α_i or τ_i). To decide the feasability of such a system from $A_k(S_P)$, we must be able, for any state s of $A_k(S_P)$, to decide whether the activation of α_i will cause a temporal fault in the future of the system. In our example, let us consider the case where the periodic system $\{\tau_1, \tau_2\}$ (see $A_1(\{\tau_1, \tau_2\})$ in Fig. 3) is in some state s. To satisfy the temporal constraints, the activation of α_1 in this context is possible only if, from s, there exists a possible behaviour of the system where there are $C_i = 2$ successive[8] time units of idleness of at least one processor for the $D_i = 5$ next time units (see Fig. 5).

To be able to decide the acceptability of α_1 in any state, we associate the transitions of the temporal model to a generating function, which is a combinatorial object whose role is to enumerate the paths corresponding to a given criterion. The generating function associated to an edge t contains the useful information to decide if α_1 can be scheduled from state $origin(t)$, where t is the first step of the scheduling sequence.

This decisional mechanism will allow us to decide if asporadic task can be scheduled without computing $A_k(S_P) \diamond A_k(S_S)$. To do this, we associate to each edge δ of $A_k(S)$ a decision function $f_\delta : \mathbb{N}^2 \to \mathbb{B}$ (where $\mathbb{B} = \{true, false\}$): $f_\delta(c_i, D_i)$ gives the feasability decision for an occurence of an α_i, with the characteristics $(C_i, D_i, any\ P_i)$, activated when the system is in the state $Origin(\delta)$. First, we present the formal tool we use, the generating series. Next, we give an algorithm to compute the extended model and we show how to use it.

Use of Generating Series to Evaluate the Idleness of Processors The behaviour of a system S is the word, i.e., a trace of a path of $A_k(S)$. Let s_t be

[8] We suppose that we cannot parallelize the tasks of the system: we can allocate a processor to α_1 during two time units which may be non successive.

the state of the system $\{\tau_1, \tau_2\}$ when α_1 is activated (see Fig. 7). Let us consider the labels $(V_{t+i})_{i\in[1,D_i]}$ of the D_i edges composing the considered path between the states s_t and s_{t+D_i}. We apply the morphism ϕ defined by

$$\left.\begin{array}{l} \begin{pmatrix} x \\ \bullet \end{pmatrix} \xrightarrow{\phi} \\ \begin{pmatrix} \bullet \\ x \end{pmatrix} \xrightarrow{\phi} \end{array}\right\} \rightarrow yx_1 \qquad \begin{array}{l} \begin{pmatrix} \bullet \\ \bullet \end{pmatrix} \xrightarrow{\phi} yx_1x_2 \\ \begin{pmatrix} x \\ x \end{pmatrix} \xrightarrow{\phi} y \end{array}$$

to each of these labels. ϕ is a morphism from $(L(A_2(\{\tau_1, \tau_2\})), .)$ in $(\mathbb{N}[y, x_1, x_2], \times)$. For example, let us consider the word

$$w = \begin{pmatrix} \bullet \\ v_1 \end{pmatrix} \begin{pmatrix} \bullet \\ \bullet \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

We get $\phi(w) = y^3x_1^2x_2$ (see the calculus mode in Fig. 6). The semantics associated with $y^ix_1^jx_2^k$ is the following:

- y^i: observation duration of i time units;
- x_1^j: one processor, at least, is inactive during j time units;
- x_2^k: two processors are inactive during k time units.

So, $\phi(w) = y^3x_1^2x_2$ means that, from the initial state of the path (s_t), we simultaneously can schedule two α_i's of relative deadline $D_i = 3$ with an execution time respectively equal to $C_i = 2$ and $C_i = 1$. Generally, a path ξ corresponding to a k processors scheduling sequence allows to decide the feasability of a sporadic task α_i (C_i, D_i, P_i) if $\phi(\xi)$ is equal to $y^L \prod_{j=1}^{k} x_j^{n_j}$ where $L \leq D_i$ and

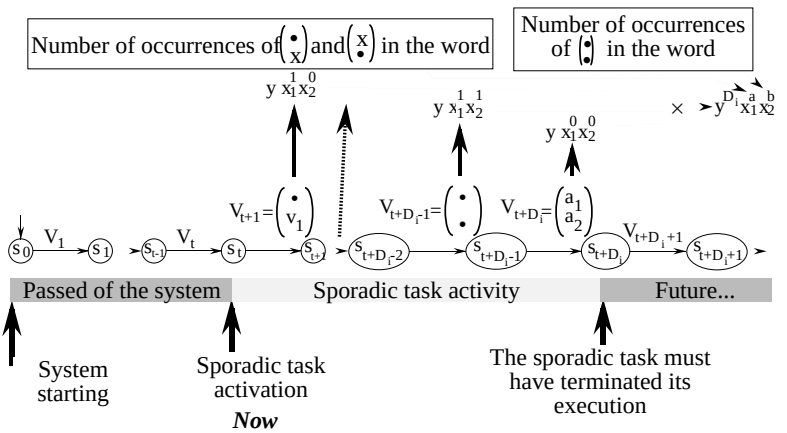


Fig. 6. Computation of the monomials from the transition labels

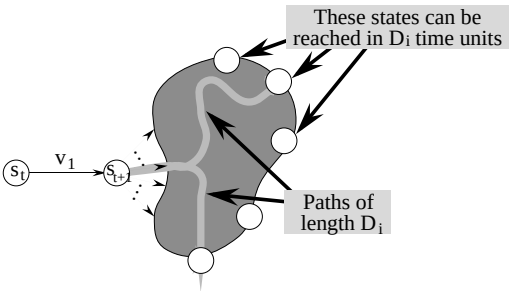


Fig. 7. Paths of length D_i where the first transition is (S_t, V_1, S_{t+1})

$\exists j \in [1,k], n_j \geq C_j$. Thus, this monomial contains the sufficient information to decide, without building $A_k(S_P) \diamond A_k(S_S)$, the scheduling of a sporadic task in a given context.

The first transition is (s_t, V_{t+1}, s_{t+1}) and is shared by many paths. So, we must adapt our monomial computing technique to a set of paths. Let us consider the case presented in Fig. 7.

Let Ξ the set of paths of length D_i starting from s_t. For each $\xi \in \Xi$, we can compute $\phi(\xi)$. Thus, we obtain $\sum_{\xi \in \Xi} \phi(\xi) = y^{D_i} \sum_{\beta=0}^{|\Xi|} \sum_{\gamma=0}^{|\Xi|} a_{\beta,\gamma} x_1^{\beta} x_2^{\gamma}$. In this polynomial, each couple (β, γ) corresponds to a configuration of idleness of the processors: the coefficient $a_{\beta,\gamma}$ enumerates the paths starting from s_t which satisfy:

- one of the two processors at least is inactive during β time units;
- the two processors are inactive during γ time units.

Example

Let us consider the state 13 of the automaton $A_2(\tau_1, \tau_2)$ (see Fig. 8). This state gets three outgoing transitions. So, when the system is in this state, three choices are compatible with the temporal constraints:

- $a_1 p_1$: τ_1 and τ_2 progress simultaneously. This possibility expresses that, in the state 13, the two tasks are active and not blocked.
- $a_1 \bullet$: τ_1 is active, but τ_2 is delayed.
- $\bullet p_1$: τ_2 is active, but τ_1 is delayed.

If α_1 is activated in the state 13, the problem consists in choosing the *good path*. So, we compute the polynomials associated with each of s-outgoing transitions (see Fig. 8) to decide of the acceptability of α_1.

The temporal characteristics of α_1 are $D_1 = 5$ and $C_1 = 2$. It can be scheduled from the state 13 if there exists, from this state, a sequence of at least 2 idle time units of at least one processor during the 5 next time units. It corresponds to the existence, in the associated polynomial, of a monomial $y^m x_1^n x_2^p$ with $n \leq D_1$ and $m \geq C_1$. The monomials of the polynomial associated with the transitions, coming from the state 13, which satisfy this criterion are represented

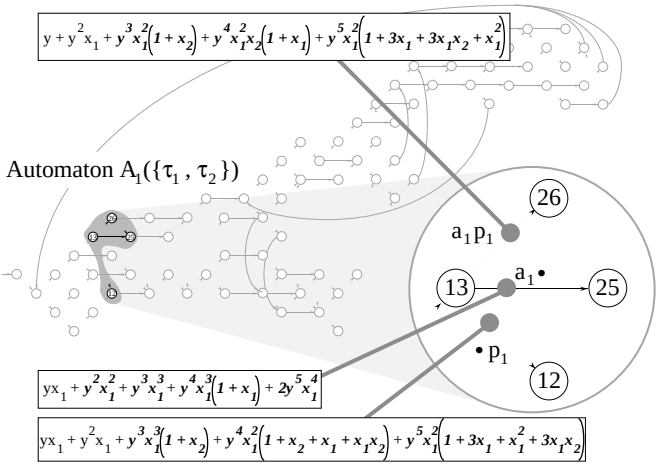


Fig. 8. Generating functions associated with each transitions

in **bold** in Fig. 8. The existence of such monomial proves the schedulability of α_1 from the state 13.

In the general case, a system S gets a finite set of periodic tasks and sporadic tasks which can be activated. Each of these tasks is characterized by a relative deadline and an execution time. Let us call D the maximal relative deadline of the set of sporadic tasks of the system. We can build $A_k(S_P)$, using the technique presented in [Gen00], and then build the automaton $D_{A_k,D}(S_P)$, where each transition is associated to the polynomial of degree D (in y), computed using the method presented below. By construction, this automaton gives an answer (among others) to the following questions:

- Can a hard sporadic task, independant of the periodic tasks, be always accepted in this system
- In a given state of the system S_P, does the occurrence of a sporadic task forbid some scheduling choice

Thus, this automaton answer to the whole problem.
In the following section, we give an algorithm to produce $D_{A_k,D}(S_P)$ from $A_k(S_P)$.

4 Implementation

In a general way, the calculus of the enumeration series associated with a language is based on the mapping, by the morphism ϕ introduced section 8, from the finite automaton to a linear system with polynomial coefficients. The translation mechanism is presented in Fig. 9. Each state i is associated with an equation of the linear system (see Fig. 9.3 and 9.4). In our case, all the states of the automaton are terminal, the equation associated with the state i is

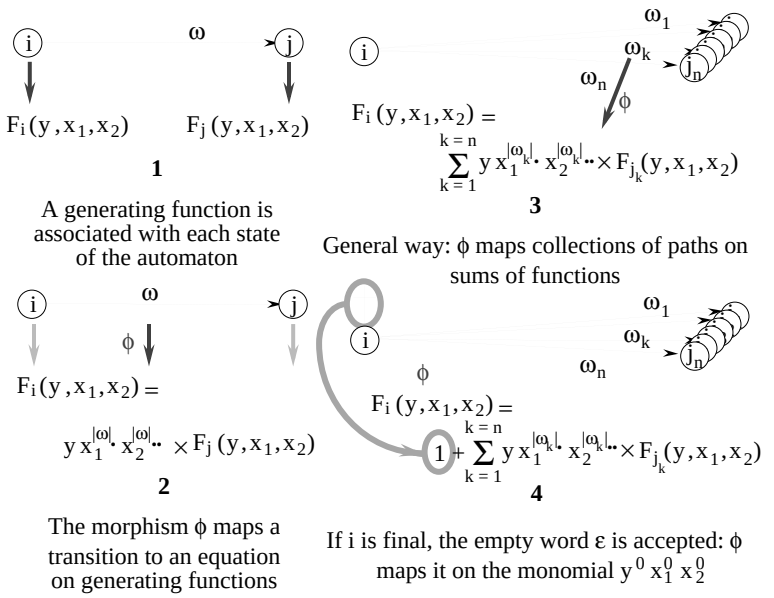


Fig. 9. From the automaton to the linear system

$F_i(y, x_1, x_2) = 1 + \sum_{j=1}^{n} M_{i,j} F_j(y, x_1, x_2)$, where $M_{i,j}$ is the polynomial (possibly null) which is the image of the edge (i, w, j) by ϕ. So, we obtain the system

$$\begin{pmatrix} M_{1,1} - 1 & \cdots & M_{n,1} \\ \vdots & & \vdots \\ M_{1,n} & \cdots & M_{n,n} - 1 \end{pmatrix} \times \begin{pmatrix} F_1 \\ \vdots \\ F_n \end{pmatrix} = \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix}$$

where n is the number of states of $A_k(S_P)$. In [CS63] it is proved that the system gets always only one solution (the vector of the F_i's). Each F_i is naturally a rational fraction with integer coefficients, which can be expanded into a series.

Once the vector $(F_i)_{i\in[1,n]}$ is known, we compute $\phi(x) \times F_j$ for each transition (i, x, j): it is a rational fraction with integer coefficients too. Thus, this fraction can be expanded into a series according to y to the needed order[9], and thus to the order D. This gives us an effective method to compute $D_{A_k,D}(S_P)$ from $A_k(S_P)$.

[9] It contains the necessary and sufficient information to decide the schedulability of a sporadic task characterized by an arbitrary large relative deadline.

5 Conclusion

We have established here several results:

- The methodology of the temporal modelisation for periodic systems, given in [Gen00], has been extended to the systems containing sporadic tasks;
- The mechanism of decision of schedulability is always valid when the system contains sporadic tasks. Concerning the system of tasks containing sporadic tasks, we have tested, at the present time, some examples of small size. However, we can propose from it, some ways of study.

The analysis of the generating functions associated with our methodology shows that they are too expressive for the needs of our study: in fact the answer to the following question:

Does it exist a valid scheduling?

corresponds to the property:

There exists at least one monomial which satisfies the corresponding temporal property.

So, our present experimentations turn towards the definition of an expressiveless class of polynomials which, we hope, will be easier to compute (in terms of complexity, as good in time as in memory). In the middle run, our objective is to obtain a methodology which could be exploited in the study of real cases. Our preoccupations turn towards, on the one hand, the improvement of the cost of the decision calculus and, on the other hand, to the definition of the class(es) of minimal functions (in term of cost) adapted to the decision problem connected to the schedulability of a real-time system.

In a long run, our objective is to use the generating series for statistical analysis of systems:

- evaluation of the proportion of accepted ingoing events according to the software (and hardware) configuration,
- study of the correlations between the states of the system and the acceptance of sporadic tasks,
- help to the determination of temporal parameters (such as the pseudo period P_i for example) which allows to schedule a configuration with given acceptation level for sporadic tasks.

References

[ABBL98] L. Aceto, P. Bouyer, A. Burgue, and K. G. Larsen. The power of reachability testing for timed automata. In *Proc. of* 18^{th} *Conf. Found. of Software Technology and Theor. Comp. Sci.*, LNCS 1530, pages 245–256. Springer-Verlag, December 1998. 89

[ABD+95] N. C. Audsley, A. Burns, R. I. David, K. W. Tindell, and A. J. Welling. Fixed priority preemptive scheduling : an historical perspective. *The journal of Real-Time Systems*, 8:173–198, 1995. 88

[Arn94] A. Arnold. *Finite transition systems.* Prentice Hall, 1994. 91

[Bak91] T. P. Baker. Stack-based scheduling of real-time processes. *the Journal of Real-Time Systems*, 3:67–99, 1991. 88

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997. 88

[CS63] N. Chomsky and M. P. Schützenberger. The algebraic theory of context-free languages. *Computer Programming and Formal Systems*, pages 118–161, 1963. 98

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University Eindhoven, 1965. 90

[Eil76] S. Eilenberg. *Automata Languages and machines*, volume A. Academic Press, 1976. 90

[GCG00] E. Grolleau and A. Choquet-Geniet. Scheduling real-time systems by means of petri nets. In *Proc. of* $25^{t}h$ *Workshop on Real-Time Programming*, pages 95–100. Universidad Politécnica de Valencia, 2000. 89

[Gen00] D. Geniet. Validation d'applications temps-réel à contraintes strictes à l'aide de langages rationnels. In *RTS'2000*, pages 91–106. Teknea, 2000. 88, 90, 93, 97, 99

[Gro99] E. Grolleau. *Ordonnancement Temps-Réel Hors-Ligne Optimal à l'Aide de Réseaux de Petri en Environnement Monoprocesseur et Multiprocesseur.* PhD thesis, Univ. Poitiers, 1999. 88, 89

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973. 88

[LM80] J. Y. T. Leung and M. L. Merill. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980. 88

[Mok83] A. K. Mok. *Fundamental Design Problems for the Hard Real-Time Environments.* PhD thesis, MIT, 1983. 88

[SCE90] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. In *Proc. of SPDS'90*, pages 578–585, 1990. 88

[SSRB98] A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo. *Deadline Scheduling for Real-Time Systems.* Kluwer Academic Press, 1998. 87

Bounded-Graph Construction for Noncanonical Discriminating-Reverse Parsers

Jacques Farré[1] and José Fortes Gálvez[2]

[1] Laboratoire I3S, CNRS and Université de Nice - Sophia Antipolis
[2] Depart. de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria

Abstract. We present a new approach for the construction of NDR parsers, which defines a new form of items and keeps track of bounded sequences of subgraph connections. This improves the precise recovery of conflicts' right-hand contexts over the basic looping approach, and thus allows to extend the class of accepted grammars. Acceptance of at least all LALR(k) grammars, for a given k, is guaranteed. Moreover, the construction needs no subgraph copies. Since bounded-graph and basic looping constructions only differ in the accuracy of the conflicts' right-hand contexts computation, the NDR parsing algorithm remains unchanged.

1 Introduction

Discriminating-reverse, DR(k), parsers [5, 7] are shift-reduce parsers that use a plain-symbol parsing stack. They decide next parsing action with the help of a DFA exploring a minimal stack suffix from the stack top, typically less than two symbols on average [6]. DR parsing is deterministic and linear on the input length [8]. DR(k) parsers accept the class of LR(k) grammars, and are practically as efficient as direct LR parsers, whereas they typically use very small tables in comparison with LR(k) [6].

Noncanonical discriminating-reverse, NDR, parsers extend DR with a conflict resolution mechanism. In case of conflict amongst several parsing actions, an initial *mark* is "pushed" at the stack top position. Next symbol is shifted and DR parsing resumes normally as far as subsequent actions are decided on stack suffixes that do not span beyond the mark position. Then, depending on the grammar, new mark positions may be introduced, until the input read and the left context encoded in the topmost mark allow to resolve the conflict. The resulting action is (noncanonically) performed at the initial mark position (which becomes the effective parsing top), and locally-canonical DR parsing resumes.

At construction time, marks are associated sets of *mark items*, which can be seen as nodes in a *mark-item graph*. Transitions in this graph are guided by transitions in the underlying graph of items in the DR automaton's item sets. Here we allow to improve the recognition capability over a previous basic-looping construction [3], by including a memory of at most h subgraphs connections into mark items. As previously, the new construction either guarantees DR conflict resolution or rejects the grammar.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 101–114, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Notation. We shall follow in general the usual conventions, as those in [10]. We shall 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 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 symbol not in V'; by convention, $\hat{\varepsilon} \Rightarrow \varepsilon$. Symbols in $\hat{V}' = V' \cup \{\hat{\varepsilon}\}$ are noted $\hat{X}$, and strings in $\hat{V}'^*$ are noted $\hat{\alpha}$.

We shall use ς to note a *core* $A \rightarrow \alpha \bullet \beta$. *Dotted* cores $\bullet A \rightarrow \alpha \bullet \beta$ or $A \bullet \rightarrow \alpha \bullet \beta$ will be respectively noted $\bullet\varsigma$ and $\varsigma\bullet$. We shall write $\dot{A} \rightarrow \alpha \bullet \beta$ or $\dot{\varsigma}$ when this dot position is unspecified, or remains unchanged in some computation.

2 The ς-DR(0) Automaton and Item Graph

Construction of the ς-DR(0) automaton shown here follows the DR(0) automaton construction described in [3], with the introduction of a minor change in DR items that enhances the power of the noncanonical extension presented in Sect. 3. We shall only discuss this change, and briefly present the ς-DR(0) construction.

2.1 ς-DR(0) Items

A ς-DR(0) item ι has the form $[\varsigma, \dot{\varsigma}']$, while original DR(0) items have the form $[i, \varsigma]$ where i denotes the parsing action.[1] As in original items, core $\varsigma = A \rightarrow \alpha \bullet \beta$ indicates that next stack symbols to explore are those in α from right to left (see Fig. 1, where $\sigma\gamma$ is the stack suffix already explored), and then those in all legal stack prefixes τ on the left of A.

For dotted core $\dot{\varsigma}'$, $\varsigma' = B \rightarrow \gamma \bullet \varphi$ is the core of the item in the kernel set I^0 (defined below), which produces ι through states transitions and closures. The rationale for this change is that there can exist more than one item for shift actions in I^0 which produce, after closures and transitions, items with a same core $A \rightarrow \alpha \bullet \beta$. In the original DR construction, this results in merging states corresponding to distinct contexts for shift actions. The new item form guarantees that descendants of distinct kernel items cannot be equal, and thus preserves from merging states. Since this introduces a new differentiation only amongst items for a shift, the cost to pay is, in most cases, a null or very small increase in the number of states.

For a reduction $B \rightarrow \gamma$, $\varsigma'\bullet = B \bullet \rightarrow \gamma \bullet$ indicates that γ has not been fully explored. Otherwise, we have $\bullet\varsigma' = \bullet B \rightarrow \gamma \bullet$. By convention, $\dot{\varsigma}' = \bullet\varsigma'$ for shift actions. The parsing action can easily be deduced from $\dot{\varsigma}'$:

$$p(\dot{\varsigma}') = \begin{cases} i & \text{if } \varsigma' = B \rightarrow \gamma \bullet \text{ and } B \xrightarrow{i} \gamma \\ 0 & \text{if } \varsigma' = B \rightarrow \gamma \bullet a\ \psi. \end{cases}$$

[1] By convention, reductions are coded by the rule number $i > 0$, and shifts by 0.

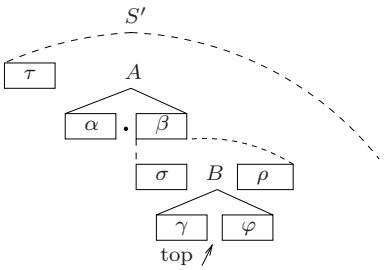


Fig. 1. Tree for ς-DR(0) item $[A{\rightarrow}\alpha{\bullet}\beta, \dot{\varsigma}']$, with $\dot{\varsigma}' = \dot{B}{\rightarrow}\gamma{\bullet}\varphi$

2.2 ς-DR(0) Initial State and Transition Function

We briefly present the new construction, since it is very close to the original one. To each ς-DR(0) automaton state q is associated a set I_q of ς-DR(0) items, i.e., $I_{q'} = I_q$ implies $q' = q$.

The closure of an item set I_q is the minimal set Δ_0 such that

$$\Delta_0(I_q) = I_q \cup \{[A{\rightarrow}\alpha{\bullet}C\beta, {\bullet}\varsigma] \mid A \rightarrow\alpha C\beta \in P', [C{\rightarrow}{\bullet}\sigma, \dot{\varsigma}] \in \Delta_0(I_q)\}.$$

The item set for initial state is computed from its kernel I^0 as follows:

$$\begin{aligned} I^0 &= \{[A{\rightarrow}\alpha{\bullet}, A{\bullet}{\rightarrow}\alpha{\bullet}] \mid A{\rightarrow}\alpha \in P'\} \\ &\quad \cup \{B{\rightarrow}\beta{\bullet}a\ \gamma, {\bullet}B{\rightarrow}\beta{\bullet}a\ \gamma] \mid B{\rightarrow}\beta a\ \gamma \in P'\}, \\ I_{q_0} &= \Delta_0(I^0). \end{aligned}$$

Last, the transition function is:

$$\Delta(I_q, X) = \Delta_0(\{[A{\rightarrow}\alpha{\bullet}X\beta, \dot{\varsigma} \mid [A{\rightarrow}\alpha X{\bullet}\beta, \dot{\varsigma}] \in I_q\}).$$

It is useful to extend the transition function to strings in V'^*:

$$\Delta(I_q, \varepsilon) = I_q, \qquad \Delta(I_q, X\alpha) = \Delta(\Delta(I_q, \alpha), X).$$

A *node* $\nu = [\varsigma, \dot{\varsigma}']_q$ is defined by its item $[\varsigma, \dot{\varsigma}']$ and item-set I_q. In the following, $\exists\,[A{\rightarrow}\alpha{\bullet}\beta, \dot{\varsigma}]_q$ will stand for $\exists\,[A{\rightarrow}\alpha{\bullet}\beta, \dot{\varsigma}] \in I_q$.

2.3 Parsing Decisions on a Stack Symbol

We briefly recall actions (or transition) taken in state q on stack symbol X. As noted, $\dot{\varsigma}$ plays the role of dotted action i in the previous construction, i.e., if $i = p(\varsigma)$, ${\bullet}\varsigma$ and $\varsigma{\bullet}$ correspond to ${\bullet}i$ and $i{\bullet}$, respectively.

As defined in next section, mark m_0^{qX} to push will be associated some mark item set $J_{m_0^{qX}}$.

$$\mathrm{At}(q,X) = \begin{cases} \text{error} & \text{if } \nexists\,[A{\to}\alpha X \bullet \beta, \dot{\varsigma}]_q \\ \text{sh/red } i & \text{else if } \nexists [A{\to}\alpha Y X \bullet \beta, \varsigma \bullet]_q \text{ and } (\forall [A{\to}\alpha X \bullet \beta, \dot{\varsigma}]_q, p(\varsigma) = i) \\ \text{push } m_0^{qX} & \text{else if } \nexists\,[A{\to}\alpha Y X \bullet \beta, \varsigma \bullet]_q \text{ and} \\ & \quad (\nexists\,[A{\to}\alpha X \bullet \beta, \dot{\varsigma}]_q, [A'{\to}\alpha' X \bullet \beta', \dot{\varsigma}']_q, p(\varsigma) \neq p(\varsigma'), A\alpha \neq A'\alpha') \\ \text{goto } q' & \text{otherwise, where } I_{q'} = \Delta(I_q, X). \end{cases}$$

As previously, the construction is progressive, i.e., only states involved in "goto's" effectively exist, in addition to states for ε-deriving stack suffixes (needed for the ε-skip connection shown below). Accordingly, for the transition function of next section, only these states (and their corresponding items) are considered.

2.4 Node Transition Function

Let $\bar{\Delta}$ be the effective state transition function, as just noted. (Single) transitions are defined for nodes on $\hat{V}'$:

$$\delta([A{\to}\alpha \bullet \beta, \dot{\varsigma}]_q, \hat{X}) = \begin{cases} \{[A{\to}\alpha' \bullet \hat{X}\beta, \dot{\varsigma}]_{q'} \mid I_{q'} = \bar{\Delta}(I_q, \hat{X})\} & \text{if } \alpha = \alpha'\hat{X} \\ \{[B{\to}\gamma \bullet A\ \varphi, \bullet\varsigma]_q\} & \text{if } \alpha = \varepsilon \text{ and } \hat{X} = \hat{\varepsilon} \\ \emptyset & \text{otherwise.} \end{cases}$$

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.

In the following, $\nu' \in \delta(\nu, \hat{\alpha})$ will also be noted $\nu' \overset{\hat{\alpha}}{\leftarrow} \nu$.

3 The Bounded-Graph Solution

In order to determine in which states a mark can be encountered, we need to compute the mark positions in the derivation trees compatible with the conflict context. These positions are defined by *mark items*.

Connected components of the ς-DR(0) item graph encode pruned derivation trees, and allow walks along left-hand side of these trees. This graph can be used to guide transitions in the mark-item graph, i.e., right-hand side walks. Mark-item transitions need to add, in the general case, an unbounded number of extra connections to the mark-item graph, and some form of looping must be devised.

In the basic looping solution presented in [3], extra transitions are added by a *connect* procedure to the basic mark-item subgraphs for each distinct conflict context. A possible way to implement these extra transitions is to build actual copies of mark items, resulting in distinct mark-item subgraphs for the different conflicts.

We present here a different, *bounded-graph* approach, where extra transitions are coded by *context sequences* κ. These sequences consist of at most h node pairs $(\nu_t, \nu_a)_L$, which guide transitions in mark-item subgraphs that are entered and exited in reverse order. These transitions are restricted to the corresponding paths $\hat{\rho}$ between ν_a and ν_t such that $\hat{\rho} \Rightarrow^+ x \in L$. Thus, differently from the basic looping approach, no mark item copying is necessary.

This bounded-graph construction permits a precise context computation of at least h graph connections. In the presentation that follows, after resuming $|\kappa|$, $|\kappa| \leq h$, graphs, the context sequence becomes empty. We note ε the null context sequence, which allows to follow any context allowed by the grammar.

Since, in basic looping, contexts added by extra transitions are restricted, this may result in some cases in a computation more precise than in the bounded-graph approach when in any-context. Consequently, the parsing power of both methods are incomparable.[2]

3.1 Mark Items

A mark item μ takes the general form $[j, \kappa, \varsigma]$, for some action j in DR(0) conflict. A mark m is associated at construction time a set J_m of mark items, and $J_m = J_{m'}$ implies $m = m'$. Since each mark-item component belongs to a finite set, the set of marks is finite.

The dot position in $\varsigma = A \rightarrow \alpha \bullet \beta$ in each μ in J_m corresponds to the stack top at the moment of "pushing" mark m. Mark-item transitions move this dot from left to right. When the right end of the rightpart is reached, the dot ascends in the parsing trees according to the encoded context (function $\hat{\theta}$ shown below). Accordingly, we define the following mark-item transition function:

$$\theta([j, \kappa, A \rightarrow \alpha \bullet \beta], \hat{X}) = \begin{cases} \{[j, \kappa, A \rightarrow \alpha \hat{X} \bullet \beta']\} & \text{if } \beta = \hat{X}\beta' \\ \hat{\theta}([j, \kappa, A \rightarrow \alpha \bullet]) & \text{if } \beta = \varepsilon \text{ and } \hat{X} = \hat{\varepsilon} \\ \emptyset & \text{otherwise,} \end{cases}$$

where $\hat{\theta}$ is defined as follows:

$$\hat{\theta}([j, \kappa, A \rightarrow \alpha \bullet]) = \begin{cases} \{[j, \kappa'\kappa_1', B \rightarrow \gamma A \bullet \gamma'] \mid \\ \quad \nu_a' = [B \rightarrow \bullet \gamma A\ \gamma', \varsigma]_q \overset{\gamma}{\leftarrow} \nu_1 \overset{\hat{\varepsilon}}{\leftarrow} \nu_a,\ \nu_t \overset{\hat{\rho}}{\leftarrow} \nu_1,\ \hat{\rho} \Rightarrow^+ x \in L, \\ \quad (\gamma = \gamma_1\hat{\rho},\ \kappa_1' = \varepsilon) \text{ or } (\hat{\rho} = \hat{\rho}_1\hat{\varepsilon}\gamma,\ \kappa_1' = (\nu_t, \nu_a')_L)\} & \text{if } \kappa = \kappa'(\nu_t, \nu_a)_L \\ \{[j, \varepsilon, B \rightarrow \gamma A \bullet \gamma'] \mid B \rightarrow \gamma A\ \gamma' \in P'\} & \text{if } \kappa = \varepsilon. \end{cases}$$

[2] In fact, it is possible to combine both approaches.

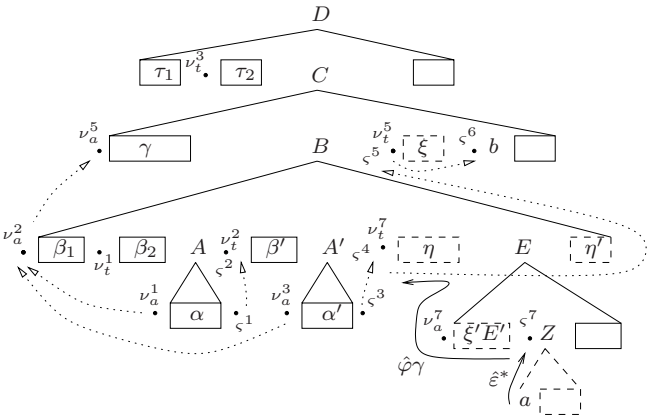


Fig. 2. Illustration of subgraph connection and context recovery

Ascent performed by $\hat{\theta}$ may be guided by the rightmost context-sequence subgraph (first case), or, in the case of a null context, any possibility allowed by the grammar is followed (second case).

Guided ascent follows a subgraph as long as its "top" ν_t is not reached while in the path on γ, or it switches to the previous subgraph κ', which may be null. In both cases, ascent may be restricted to subgraphs connected from an ε-skip (see Sect. 3.3), in which case $L = T^0 = \{\varepsilon\}$. See for instance Fig. 2, where both $\hat{\theta}([j, (\nu_t^3, \nu_a^1)_L, \varsigma^1])$ and $\hat{\theta}([j, (\nu_t^3, \nu_a^2)_L (\nu_t^1, \nu_a^1)_{L'}, \varsigma^1])$ may contain $[j, (\nu_t^3, \nu_a^2)_L, \varsigma^2]$, provided that in the former case $\hat{\rho} = \tau_2 \hat{\varepsilon} \gamma \hat{\varepsilon} \beta_1 \beta_2 \Rightarrow^+ x \in L$, and in the latter case $\hat{\rho} = \beta_2 \Rightarrow^+ x' \in L'$. Accordingly, $\hat{\theta}([j, (\nu_t^1, \nu_a^1)_{L'}, \varsigma^1])$ would contain $[j, \varepsilon, \varsigma^2]$.

We extend θ to strings in $\hat{V}'^*$:

$$\theta(\mu, \varepsilon) = \{\mu\}, \qquad \theta(\mu, \hat{X}\hat{\alpha}) = \bigcup_{\mu' \in \theta(\mu, \hat{X})} \theta(\mu', \hat{\alpha}).$$

In the following, $\mu' \in \theta(\mu, \hat{\alpha})$ will also be noted $\mu \xrightarrow{\hat{\alpha}} \mu'$.

3.2 Connection Function

This function connects, if necessary, the subgraph for final node ν_t to the context of a mark item μ. Only paths from the starting node $[\varsigma', \dot{\varsigma}_0]_{q_0}$ to ν_t producing some string x in language L are considered. When this subgraph is associated to a reduction, the mark position is set at $\varsigma' = \varsigma_0 = A \rightarrow \alpha \bullet$. In the case of a shift mark positions are set at $\varsigma' = \varsigma_1 = A \rightarrow \gamma_1 X \bullet Y \sigma$.

$$C_L(\mu, \nu_t) =$$
$$\{[j, \kappa\kappa_1, \varsigma'] \mid \mu = [j, \kappa, \varsigma], [\varsigma, \dot{\varsigma}_0]_q = \nu_t \xleftarrow{\hat{\varphi}} \nu_a \xleftarrow{\gamma} [\varsigma', \dot{\varsigma}_0]_{q_0},\ \hat{\varphi}\gamma \Rightarrow^* x \in L,$$
$$(\hat{\varphi} \in V^*,\ \kappa_1 = \varepsilon) \text{ or } (\hat{\varphi} = \hat{\varphi}_1 \hat{\varepsilon},\ \kappa_1 = (\nu_t, \nu_a)_L),$$
$$\nu_2 \xleftarrow{X} [\varsigma_1, \dot{\varsigma}_0]_{q_0} \xleftarrow{\hat{\varepsilon}^*} [\varsigma_0, \dot{\varsigma}_0]_{q_0},\ (p(\varsigma_0) = 0,\ \varsigma' = \varsigma_1) \text{ or } (p(\varsigma_0) > 0,\ \varsigma' = \varsigma_0)\}.$$

Note that no new pair is added ($\kappa_1 = \varepsilon$) when the subgraph to connect would simply move the mark position along the same rightpart in the derivation tree. Referring to Fig. 2, $[j, (\nu_t^3, \nu_a^2)_{L}(\nu_t^2, \nu_a^3)_{T^*}, \varsigma^3]$.

3.3 ε-Skip Function

Since, after pushing a conflict mark, next terminal will be shifted, mark items must represent positions on the left of this terminal in the derivation trees. The shift ensures that marks are separated in the parsing stack by at least one symbol deriving some non-empty section of the input string. Thus, parsing will not indefinitely push marks.

Reaching a position just to the left of some terminal may imply to skip sequences of ε-deriving nonterminals. First, an ascending walk on the right-hand side of the tree may be performed, giving positions on left symbols deriving a non empty terminal sequence. Then, a left-hand side descending walk may be needed, which will perform a graph connection. Thus, the ε-skip function is defined as follows:

$$\theta_\varepsilon(\mu) = \{\mu'' \mid \mu \xrightarrow{\hat{\rho}\hat{\varepsilon}} \mu' = [j, \kappa, \varsigma], \hat{\rho} \Rightarrow^* \varepsilon,\ \mu'' \in C_{T^0}(\mu', [\varsigma, \dot{\varsigma}_0]_q),\ p(\varsigma_0) = 0\}.$$

Ascent through $\hat{\rho}$ by θ_ε on $[j, (\nu_t^3, \nu_a^2)_{L}(\nu_t^2, \nu_a^3)_{T^*}, \varsigma^3]$ produces $[j, (\nu_t^3, \nu_a^2)_{L}, \varsigma^4]$ and, if $\eta E \eta' \Rightarrow^* \varepsilon$, $[j, (\nu_t^3, \nu_a^5)_{L}, \varsigma^5]$, which correspond to cases $\hat{\rho} = \varepsilon$ and $\hat{\rho} = \hat{\varepsilon}\eta E \eta'$, respectively. The connection by C_{T^0} will produce in the former case $[j, (\nu_t^3, \nu_a^2)_{L}(\nu_t^7, \nu_a^7)_{T^0}, \varsigma^7]$, provided that $\eta \xi' E' \Rightarrow^* \varepsilon$, and it will produce in the latter case $[j, (\nu_t^3, \nu_a^5)_{L}, \varsigma^6]$, if $\xi \Rightarrow^* \varepsilon$.

3.4 Transitions for a Mark on a State

For each state q in which a mark m may give place to another, "induced", mark m', the mark-item set for mark m' is computed. This computation has the form of a transition function for mark m on state q.

Connections of the subgraphs for actions in conflict are first performed, if necessary. Then, for reductions in conflict, an ε-skip occurs, possibly involving a second connection. Finally, the context sequences are truncated to their h rightmost subgraphs.

$$\begin{aligned}\Theta(J_m, q) = \{[j, \kappa' : h, \varsigma'] \mid \mu = [j, \kappa, \varsigma] \in J_m,\ \mu' \in C_{T^*}(\mu, [\varsigma, \dot{\varsigma}_0]_q),\\ (p(\varsigma_0) = 0, [j, \kappa', \varsigma'] = \mu') \text{ or } (p(\varsigma_0) > 0,\ [j, \kappa', \varsigma'] \in \theta_\varepsilon(\mu'))\}.\end{aligned}$$

Referring again to Fig. 2, if $J_m = \{[j, (\nu_t^3, \nu_a^2)_{L}, \varsigma^2]\}$ and $I_q = \bar{\Delta}(I_{q_0}, \beta'\alpha')$, $[j, (\nu_t^3, \nu_a^2)_{L}(\nu_t^7, \nu_a^7)_{T^0}, \varsigma^7]$ as well as $[j, (\nu_t^3, \nu_a^5)_{L}, \varsigma^6]$, as we have seen in Sect. 3.3.

3.5 Initial Set of a ς-DR(0)-conflict Mark

Let m_0^{qX} be the original mark associated with some conflict (q, X). Its associated set of mark items is the following:

$$\begin{aligned}J_{m_0^{qX}} = \{[j, \kappa : h, \varsigma] \mid \nu = [\varsigma_1, \dot{\varsigma}_0]_q,\ \varsigma_1 = A \to \alpha X \bullet \beta, j = p(\varsigma_0),\\ \mu \in C_{T^*}([j, \varepsilon, \varsigma_1], \nu),\ (j = 0, \mu = [j, \kappa, \varsigma]) \text{ or } (j > 0,\ [j, \kappa, \varsigma] \in \theta_\varepsilon(\mu))\}.\end{aligned}$$

That is, a subgraph for an action in conflict is "connected" to a graph whose transitions follow all the upwards paths allowed by the grammar.

3.6 Inadequacy Condition

A grammar G is inadequate iff

$$\exists\, [j, \kappa, A{\to}\alpha\bullet\beta], [j', \kappa', A{\to}\alpha'\bullet\beta] \in J_m, j \neq j', \kappa \in \{\kappa', \varepsilon\}.$$

Since, from an item with context sequence ε, all legal paths allowed by the grammar can be followed, there is a path in each respective mark graph of each action which follows the same transitions.[3] Consequently, if the condition holds, there are some sentences for which the parser cannot discriminate between both such actions, and the grammar is rejected.

3.7 Parsing Decisions on a Mark

We say that some mark item $[j, \kappa, \varsigma]$ and some state item $[\varsigma', \dot{\varsigma}_0]$ *match*, if $\varsigma = \varsigma'$.

A DR(0) conflict can be resolved when encountering m in q if all items of m matching items of q have a same action in conflict j. And m can decide the parsing action i in q if all items of q matching items of m have a same action i. Thus, decisions in state q (see Sect. 2.3) are extended for mark m as follows:

$$\mathrm{At}(q, m) = \begin{cases} \text{error} & \text{if } \chi(q,m) = \emptyset \\ \text{resolve } j' & \text{else if } \forall (i,j) \in \chi(q,m), j = j' \\ \text{sh/red } i' & \text{else if } (\forall (i,j) \in \chi(q,m), i = i') \text{ and } (\nexists\, [A{\to}\alpha X \bullet \beta', \varsigma\bullet]_q, p(\varsigma) = i') \\ \text{push } m' & \text{otherwise, where } J_{m'} = \Theta(J_m, q), \end{cases}$$

with $\chi(q,m) = \{(i,j) \mid i = p(\dot{\varsigma}_0), \exists\, [\varsigma, \dot{\varsigma}_0] \in I_q, [j, \kappa, \varsigma] \in J_m, q \neq q_0\}$. Since one shift is performed after pushing a mark, no mark can be encountered in q_0 at parsing time.

4 BG(h)DR(0) Grammars and Languages

All BG(h)DR(0) grammars are unambiguous. They include all LALR(k) grammars (for $h = 2k$), subsets of other LALR and LRR grammars, and, finally, a subset of non-LRR grammars for a family of languages including subsets of LRR and non-LRR nondeterministic languages. Let us justify now the most important results out of these.

[3] Clearly, all paths will be the same if $\kappa = \kappa'$. The grammar is not necessarily ambiguous, even if both come from context sequences that have not been truncated. When DR(0) states are merged, distinct left contexts are mixed, and so their associated right contexts. A more powerful construction could keep on trying to discriminate on right contexts while $\kappa \neq \kappa'$, and then try to apply a process similar to [9].

4.1 LALR(k) Grammars

In LALR(k) grammars, for each LR(0) conflict state, the sets of lookaheads of length k which are compatible with any left context associated to each LR(0) item in conflict are disjoint.

Our right graph construction based on mark-item transition function θ is designed to precisely follow the right contexts which are compatible with the corresponding left paths on δ. Bound h thus guarantees that, as far as the number of right subgraph connections is not greather than h, the original conflict path can be precisely resumed.

Each transition with Θ implies at most two subgraphs connections (one by Θ itself, the other by θ_ε), while at least one terminal will be shifted. Therefore, our method precisely "computes" lookaheads of at least k symbols for $h = 2k$, in accordance with the ς-DR(0) conflict left context. The critical point is thus whether this ς-DR(0)-conflict context is at least as precise as the LR(0)-conflict context.

LR(0) conflict states are different if their items sets are, but these correspond to ς-DR(0) initial state items. Since ς-DR(0) states are different when their ς sets are, the ς-DR(0) conflict left contexts are at least as precise as the corresponding LR(0) ones. In conclusion, the BG($2k$)DR(0) construction shown here has a discriminatory power of at least LALR(k).

4.2 Subsets of Non-LR Grammars

The R(k)LR(0) [2] and the LAR(k) [1] methods rely on an LR(0) automaton. They use, at construction time, sequences of k LR(0) states in order to compute a regular cover of the right context, according to the items in these k states. Since, in general, some of these states may correspond to ε-reductions, these methods do not ensure that the cover is sufficiently precise for next k terminals, and acceptation of LALR(k) grammars is guaranteed only if they are free of ε-productions. We have recently developed a more powerful solution [4] which applies the ideas of bounded connection of subgraphs and ε-skip, and which also accepts LALR(k) grammars with ε-productions.

Mark computation in NDR is more discriminating than a regular cover of the right context, since, performing reductions after the conflict, it is able to "skip" context-free sublanguages in the right context. Thus BG(h)DR(0) accepts any grammar the above parsers for subsets of LRR accept, for $h = 2k$, and also grammars that any (ideal) LRR parser generator would reject.

Finally, as the example in [3] clearly shows, the method accepts grammars for (nondeterministic) non-LRR languages, i.e., languages for which there exists no grammar with some regular covering for resolving LR conflicts.

5 Illustration of BG(1)DR(0)

Consider the following grammar:

$S \xrightarrow{2} A\ a$ $\quad S \xrightarrow{3} DBa$ $\quad A \xrightarrow{4} CEa$ $\quad B \xrightarrow{5} DFa$ $\quad E \xrightarrow{6} GA\ c$ $\quad E \xrightarrow{7} b$
$C \xrightarrow{8} c$ $\quad D \xrightarrow{9} c$ $\quad F \xrightarrow{10} Bcc$ $\quad F \xrightarrow{11} b$ $\quad G \xrightarrow{12} \varepsilon$

The construction presented in previous sections finds two ς-DR(0) conflicts. A first conflict between shift and reduction 12 is found in state[4] q_ε on stack symbol C (see Fig. 3). In order to compute the corresponding mark m_0 item set, context sequences ε and $(\nu_t'^0, \nu_0'^0)_{T^*}$ are temporary obtained from C_{T^*}. After θ_ε ascent, the latter subgraph becomes $(\nu_t'^0, \nu_a'^0)_{T^*}$. Thus, for $h = 1$,

$$J_{m_0} = \{[0,\ \varepsilon,\ A \rightarrow C \bullet Ea],\ [12,\ (\nu_t'^0, \nu_a'^0)_{T^*},\ E \rightarrow G \bullet A\ c]\}.$$

Only the first or the second mark item matches some node in q_b or q_c, respectively. Accordingly, during parsing this conflict is immediately resolved in favor of shift or reduce 12 after reading b or c, respectively.

A second conflict, whose resolution needs unbounded right-hand context exploration[5], is found between reductions 8 and 9 in state q_c on the bottom-of-stack symbol $\vdash$. Starting with m_1, the following mark item sets are computed:

J_{m_1}={$[8, (\nu_t^1, \nu_a^1)_{T^*}, A \rightarrow C \bullet Ea]$, $[8, (\nu_t^\varepsilon, \nu_a^\varepsilon)_{T^0}, E \rightarrow G \bullet A\ c]$, $[9, (\nu_t'^1, \nu_a'^1)_{T^*}, S \rightarrow D \bullet Ba]$},
J_{m_2}={$[8, (\nu_t^2, \nu_a^1)_{T^*}, A \rightarrow C \bullet Ea]$, $[8, (\nu_t^\varepsilon, \nu_a^\varepsilon)_{T^0}, E \rightarrow G \bullet A\ c]$, $[9, (\nu_t'^2, \nu_a'^2)_{T^*}, B \rightarrow D \bullet Fa]$},
J_{m_3}={$[8, (\nu_t^2, \nu_a^1)_{T^*}, A \rightarrow C \bullet Ea]$, $[8, (\nu_t^\varepsilon, \nu_a^\varepsilon)_{T^0}, E \rightarrow G \bullet A\ c]$, $[9, (\nu_t'^3, \nu_a'^2)_{T^*}, B \rightarrow D \bullet Fa]$},
J_{m_4}={$[8, (\nu_t^2, \nu_a^1)_{T^*}, A \rightarrow CE \bullet a]$, $[9, (\nu_t'^2, \nu_a'^2)_{T^*}, B \rightarrow DF \bullet a]$},
J_{m_5}={$[8, (\nu_t^2, \nu_a^1)_{T^*}, A \rightarrow CE \bullet a]$, $[9, (\nu_t'^3, \nu_a'^2)_{T^*}, B \rightarrow DF \bullet a]$},
J_{m_6}={$[8, \varepsilon, E \rightarrow GA\ \bullet c]$, $[9, \varepsilon, S \rightarrow DB \bullet a]$},
J_{m_7}={$[8, \varepsilon, E \rightarrow GA\ \bullet c]$, $[9, (\nu_t'^3, \nu_a'^3)_{T^*}, F \rightarrow B \bullet cc]$},
J_{m_8}={$[8, \varepsilon, A \rightarrow CE \bullet a]$, $[9, (\nu_t'^3, \nu_a'^3)_{T^*}, F \rightarrow Bc \bullet c]$}.

Figure 4 shows the subgraphs for the corresponding context-sequence nodes.

In order to compute J_{m_1}, during θ_ε, and starting at ν_0^1 and $\nu_0'^1$, reference nodes ascend until ν_a^1 and $\nu_a'^1$, respectively, and the subsequent C_{T^0} involves subgraphs in the lower section of Fig. 4. In particular, truncation with $h = 1$ results, from $(\nu_t^1, \nu_a^1)_{T^*} (\nu_t^\varepsilon, \nu_a^\varepsilon)_{T^0} : 1$, in the context sequence for the second mark item of J_{m_1}. When the next shifted terminal is b, m_1 resolves the conflict in favor of reduction 8, since only the first item of m_1 matches some node in q_b.

Mark m_1 gives place to mark m_2 if the case of a second c, since ν_t^2 and $\nu_t'^2$ match items of m_1 associated to both actions in conflict. Graph connections and ε-skips are performed, and context sequences are truncated if necessary. In

[4] In this section we shall use the notation q_σ if $I_{q_\sigma} = \bar{\Delta}(I_{q_0}, \sigma)$, e.g., initial state shall be noted q_ε.

[5] The languages on the right of this conflict are $c^n ba\ (ca)^n a$ and $c^{n+1} ba\ (cca)^n a, n \geq 0$. Consequently, the grammar is not LR(k) for any k, although it is BG(1)DR(0) as well as LRR, as we shall see.

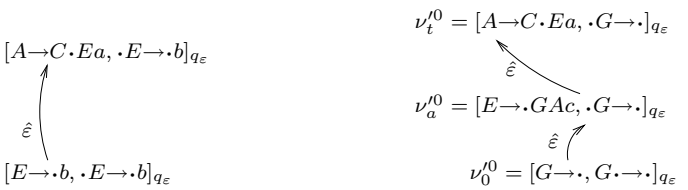


Fig. 3. Subgraphs for conflict (q_ε, C)

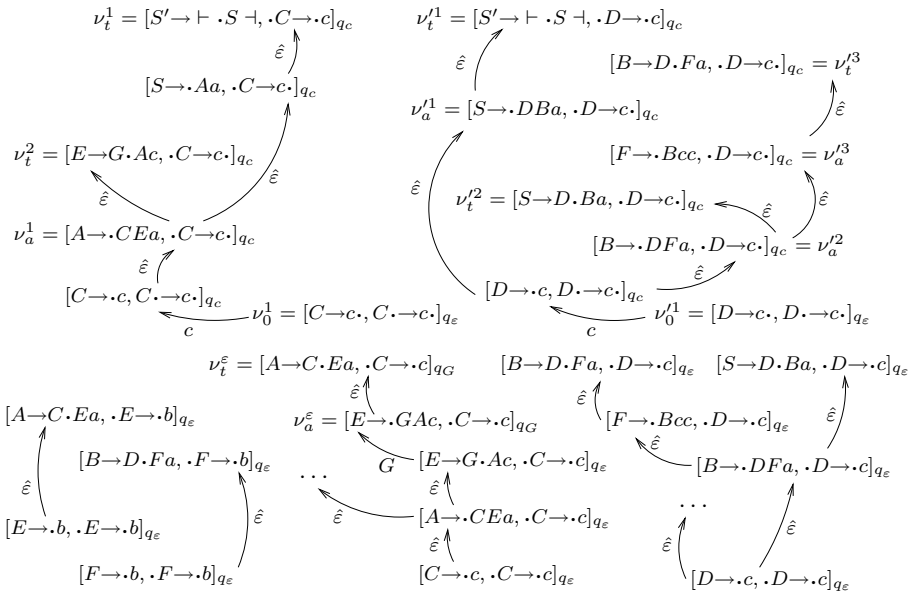


Fig. 4. Nodes in mark-item context sequences, for conflict $(q_c, \vdash)$

a similar way, mark m_2 gives place in q_c to mark m_3, which reproduces itself again in q_c, since context sequences are truncated.

Marks m_2 and m_3, in q_b, give place (see the upper subgraphs in Fig. 5) to marks m_4 and m_5, respectively; note how the correspondings core dots move rightwards. These new marks give place in q_a to marks m_6 and m_7, respectively (see middle subgraphs Fig. 5): the θ_ε ascent produces empty context sequences, except for the second item of m_7, where ν'^2_a ascends to ν'^3_a (Fig. 4.)

Finally, m_6 resolves the conflict in q_c or q_a, while m_7 has still to give place (lower subgraphs in Fig. 5) to m_8 in q_c.

The resulting mark automaton is shown in Fig. 6, from which the contexts on the right of the conflict encoded by the marks can be easily deduced, e.g., m_3 encodes cc^+. Only actions relevant to parsing are shown: in this example, although

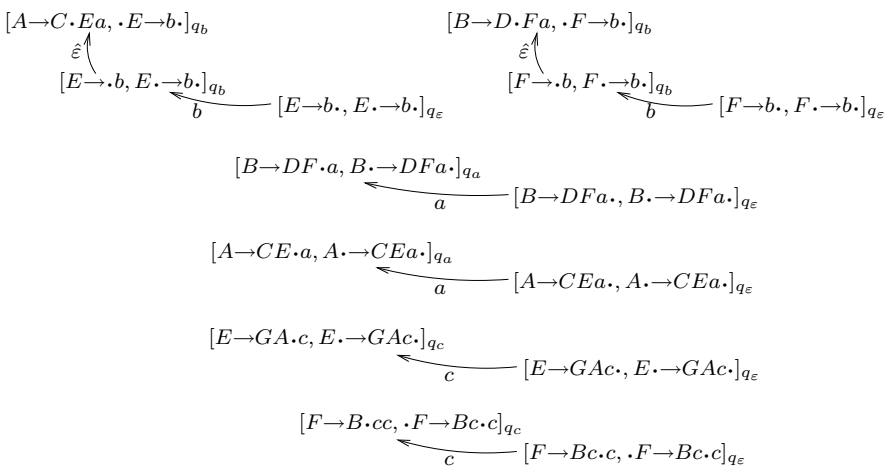


Fig. 5. Auxiliary subgraphs for conflict $(q_c, \vdash)$

no useless mark[6] is produced, some useless actions are, e.g., m_1 "resolves" the conflict in q_C and q_{Ea}, but this can never take place during parsing because C or E can only be present in the stack after the conflict is resolved.

Note, finally, that if the construction were done with $h = 0$, marks m_4 and m_5 would merge and ascend in any context, and thus it would be impossible to separate actions 8 and 9.

5.1 Parsing Example

Let see a parsing example for the sake of completion. The NDR parsing algorithm is given in [3]. Since marks are not present in stack, only positions of the ς-DR(0) conflict (noted $|$) and (conflict's rightmost) mark m_i (noted $|_i$) are shown.

According to the mark automaton of Fig. 6, the configuration of stack plus remaining input, noted *stack* ❙ *input*, would evolve as follows, for $n \geq 2$:

$$\vdash \text{❙}\ c^{n+1}ba\ (ca)^n a\ \dashv \models \vdash c\ \text{❙}\ c^n ba\ (ca)^n a\ \dashv \models \vdash c||_1 c\ \text{❙}\ c^{n-1}ba\ (ca)^n a\ \dashv \models$$
$$\vdash c|c|_2 c\ \text{❙}\ c^{n-2}ba\ (ca)^n a\ \dashv \models \vdash c|cc|_3 c\ \text{❙}\ c^{n-3}ba\ (ca)^n a\ \dashv \models \cdots \models$$
$$\vdash c|c^n|_3 b\ \text{❙}\ a(ca)^n a\ \dashv \models \vdash c|c^n b|_5 a\ \text{❙}\ (ca)^n a\ \dashv \models \vdash c|c^n ba\ |_7 c\ \text{❙}\ a(ca)^{n-1}a\ \dashv$$
$$\models$$
$$\vdash c|c^n ba\ c|_8 a\ \text{❙}\ (ca)^{n-1}a\ \dashv.$$

Now, the conflict is resolved in favor of reduction 8. The effective DR parsing top is put at the ς-DR(0) conflict point, reduction 8 takes place, and parsing resumes:

[6] As in the basic-looping construction, production of useless mark do not reduce the accepted grammar class.

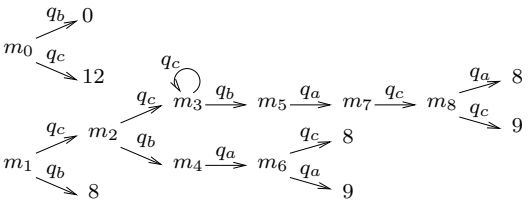


Fig. 6. Mark automaton

$\vdash c|c^n ba\ c|_8 a \shortmid (ca)^{n-1}a\ \dashv \models \vdash C \shortmid c^n ba\ (ca)^n a\ \dashv$.

A shift-reduce 12 conflict occurs now, giving mark m_0, and is immediately resolved:

$\vdash C \shortmid c^n ba(ca)^n a\ \dashv \models \vdash C||_0 c \shortmid c^{n-1}ba\ (ca)^n a\ \dashv \models \vdash CG \shortmid c^n ba\ (ca)^n a\ \dashv \models$
$\vdash CGc \shortmid c^{n-1}ba\ (ca)^n a\ \dashv \models \vdash CGC \shortmid c^{n-1}ba\ (ca)^n a\ \dashv \models \cdots \models$
$\vdash C(GC)^n||_0 b \shortmid a(ca)^n a\ \dashv \models \vdash C(GC)^n b \shortmid a(ca)^n a\ \dashv \models$
$\vdash C(GC)^n E \shortmid a(ca)^n a\ \dashv \models \vdash C(GC)^{n-1}GCEa \shortmid (ca)^n a\ \dashv \models$
$\vdash C(GC)^{n-1}GA \shortmid (ca)^n a\ \dashv \models \vdash C(GC)^{n-1}GA\ c \shortmid a(ca)^{n-1}a\ \dashv \models$
$\vdash C(GC)^{n-1}E \shortmid a(ca)^{n-1}a\ \dashv \models \cdots \models \vdash CEa \shortmid a \dashv \models \vdash A \shortmid a \dashv \models$
$\vdash A\ a \shortmid \dashv \models \vdash S \shortmid \dashv \models \vdash S \dashv \shortmid \models S' \shortmid$.

6 Conclusion

The bounded-graph construction for NDR parsers represents an improvement over a previous, basic-looping approach. A mechanism of up to h graph connections, combined with the introduction of a variant of DR items, allows to accept a wider class of grammars, including grammars for nondeterministic languages, and guarantees, if needed, all LALR(k) grammars, for $h = 2k$. The proposed construction naturally detects inadequate grammars, and produces the corresponding BGDR parsers otherwise. These parsers are almost as efficient as DR parsers, and could thus be used on applications requiring high parsing power, where ambiguity or nondeterminism during parsing is hardly acceptable, as the area of programming language processing.

References

[1] Manuel E. Bermudez and Karl M. Schimpf. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41:230–250, 1990. 109

[2] Pierre Boullier. *Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques*. PhD thesis, Université d'Orléans, France, 1984. In French. 109

[3] Jacques Farré and José Fortes Gálvez. A basis for looping extensions to discriminating-reverse parsing. In M. Daley, M. G. Eramian, and S. Yu, editors, *5th International Conference on Implementation and Applications of Automata, CIAA 2000*, pages 130–139, London, Ontario, 2000. The University of Western Ontario. 101, 102, 105, 109, 112

[4] Jacques Farré and José Fortes Gálvez. A bounded-connect construction for LR-regular parsers. In R. Wilhelm, editor, *International Conference on Compiler Construction, CC 2001*, Lecture Notes in Computer Science #2027, pages 244–258. Springer-Verlag, 2001. 109

[5] José Fortes Gálvez. Generating LR(1) parsers of small size. In *Compiler Construction. 4th Internatinal Conference, CC'92*, Lecture Notes in Computer Science #641, pages 16–29. Springer-Verlag, 1992. 101

[6] José Fortes Gálvez. Experimental results on discriminating-reverse LR(1) parsing. In Peter Fritzson, editor, *Proceedings of the Poster Session of CC'94 - International Conference on Compiler Construction*, pages 71–80. Department of Computer and Information Science, Linköping University, March 1994. Research report LiTH-IDA-R-94-11. 101

[7] José Fortes Gálvez. A practical small LR parser with action decision through minimal stack suffix scanning. In Jürgen Dassow, G. Rozenberg, and A. Salomaa, editors, *Developments in Language Theory II*, pages 460–465. World Scientific, 1996. 101

[8] José Fortes Gálvez. *A Discriminating Reverse Approach to LR(k) Parsing*. PhD thesis, Universidad de Las Palmas de Gran Canaria and Université de Nice-Sophia Antipolis, 1998. 101

[9] B. Seité. A Yacc extension for LRR grammar parsing. *Theoretical Computer Science*, 52:91–143, 1987. 108

[10] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*. Springer, 1988 and 1990. 102

Finite-State Transducer Cascade to Extract Proper Names in Texts

Nathalie Friburger and Denis Maurel

Laboratoire d'Informatique de Tours
E3i, 64 avenue Jean Portalis, 37000 Tours
{friburger,maurel}@univ-tours.fr

Abstract. This article describes a finite-state cascade for the extraction of person names in texts in French. We extract these proper names in order to categorize and to cluster texts with them. After a finite-state pre-processing (division of the text in sentences, tagging with dictionaries, etc.), a series of finite-state transducers is applied one after the other to the text and locates left and right contexts that indicates the presence of a person name. An evaluation of the results of this extraction is presented.

1 Motivation

Finite-State Automata and particularly transducers are more and more used in natural languages processing [13]. In this article, we suggest the use of a finite-state transducer cascade to locate proper names in journalistic texts. In fact, we study names because of their numerous occurrences in newspapers (about 10 % of a newspaper) [3]. Proper names have already been studied in numerous works, from the Frump system [5] to the American programs Tipster[1] and MUC[2]. These two programs evaluate systems of information extraction in texts. The Named Entity Task is a particular task of MUC : this task aims to detect and categorize named entity (like proper names) in texts.

First of all, we present some known finite-state cascades used in natural language processing. Secondly we shall explain our finite-state pre-processing of texts (division of the text in sentences, tagging with dictionaries) and how to use transducers to extract patterns and categorize them. Then we shall describe our work through a linguistic analysis of texts to create the best cascade as ossible. Finally, we shall present the results of the extraction of proper names on a 165000-word text from the French newspaper *Le Monde*, and shall discuss the main difficulties and problems to be solved.

[1] www.tipster.org
[2] http://www.muc.saic.com/

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 115–124, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Finite-State Transducer Cascades in Natural Language Processing

Finite-State Transducer Cascades have been developed for a few years to parse natural language. In this part, we quickly present three systems parsing with finite-state cascades. The advantages of transducers are their robustness, precision and speed.

Abney [1] presents a syntactic parser for texts in English or German language (Cass System). He describes the main principles of a cascade and defines a cascade as a "sequence of strata". The transducer T_i parses the text L_{i-1} and produces the text L_i. Abney says that reliable patterns are found first: he calls them "islands of certainty". The uncertain patterns are found next. In the same way [11] presents a finite-state cascade to parse Swedish, which is very close to Abney's one. The IFSP System (Incremental Finite State Parser [2], created at Xerox Research Center) is another system of cascade of transducers which has been used fo a syntaxic analysis of Spanish language [8].

Fastus [9] is a very famous system for information extraction from texts in English or Japanese, sponsored by DARPA: it is closer to the work we present here. This system parses texts into larger and larger phrases. It first finds compound nouns, locations, dates and proper names. Secondly it recognizes nominal or verbal groups, and particles. Thirdly complex noun phrases and complex verb phrases are found. The previous patterns are used to discover events and relations between them. This system was presented at the MUC evaluations for information extraction and it obtained good scores.

We present our own finite-state cascade, which finds proper names and their contexts in texts. We created this system in order to cluster journalistic texts.

3 Finite-State Pre-processing

We have chosen to use Intex system [14] to pre-process texts. Intex permits to capitalize on transducers on texts for the whole processing. Firstly we pre-process texts cutting them in sentences and tagging them with dictionaries. After that we use our own program, which completes Intex's possibilities and allows realizing a finite-state transducer cascade.

3.1 Sentences

Before applying the finite-state cascade to a text, we submit it to a finite-state pre-processing. Indeed, we cut the text into sentences [7]. A transducer describes possible ends of sentences and puts the mark $\{S\}$ between each sentence.

The main difficulties come from the dot which is a very ambiguous symbol when it is followed by an upper case letter: the dot can either be the final dot of a sentence or not. So we have found four types of ambiguities with the dot:

- In person names when they are preceded by an abbreviated form with the dot, as in "*M. Jean Dupont" (Mister Jean Dupont)*: the dot in *M.* is clearly not the end of the sentence.
- In person names too when they contain an abbreviated first name as in "*J. Dupont*".
- In abbreviations such as "*N.A.T.O*".
- In different abbreviated words as in "*Éd. Gallimard*" or in "*Chap. 5*" for example.

Therefore the resolution of these ambiguities induces errors to be taken into account. For example, dots after a symbol (money, physical and chemical symbols, etc.) as in "*Ce livre coûte 20 F. Le vendeur me l'a dit.*" (*This book costs 20 F. The salesman said it to me.*) or dots after a compound word as in "*Cet aliment contient de la vitamine A. Le docteur conseille d'en manger.*" (*This food contains vitamin A. The doctor advises to eat it.*) really notify the end of a sentence.

Figure 1 presents the transducer that inserts the output $\{S\}$ between each sentence. The various cases are handled respectively in sub-graphs (grey boxes) *cas2* (person names and abbreviation patterns),*cas3* (symbols and compound words) and *cas4* (for abbreviated words).

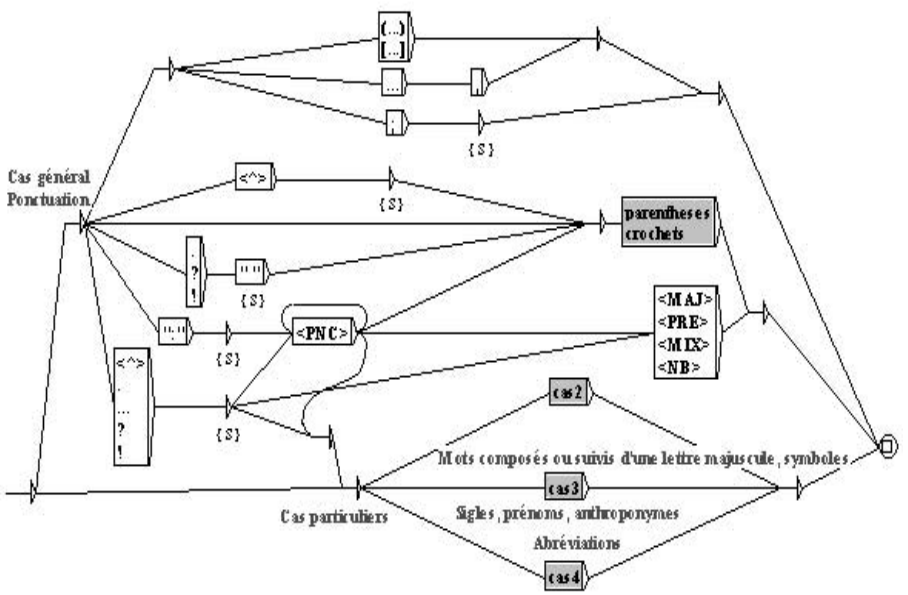


Fig. 1. Transducer describing end of sentences and ambiguous dot patterns

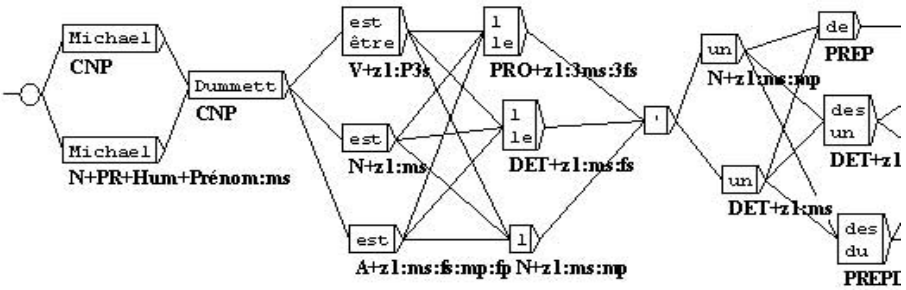


Fig. 2. A tagged sentence with Intex System

3.2 Tagging

Now we tag the text from a morpho-syntactic point of view. Thus we use dictionaries that link words with information: lemmas, grammatical categories (noun, verb, etc.) and semantic features (concrete, place-names, first names, abbreviations, etc.)[3]. The advantage to these dictionaries is double:

- Every word is given with its lemmatized form, which avoid to describe all the flexions of a word in the transducers that discover them.
- The used dictionaries contain syntactical information that can help locating patterns for proper names.

Each word is tagged with all its occurrences in dictionaries. Figure 2 shows the transducer for the beginning of the sentence "*Michael Dummett est l'un des plus grands philosophes britanniques d'aujourd'hui*" (*Michael Dummett is one of the most famous contemporary British philosophers*). This sentence is tagged with Intex and our dictionaries: the inputs are in boxes (the second line being the lemma of the word), the outputs are in bold face and contain syntactic information (N = noun, V = Verb, etc.) and semantic information (Hum = Human).

4 Finite-State Transducer Cascade: The Example for Extracting Person's Names

4.1 Transducers

Transducers are finite-state machines with an input alphabet and an output alphabet: this property can be used to extract patterns and categorize them.

[3] Delaf dictionaries of simple words and their inflected forms [4], Prolintex dictionary of place-names realized within the Prolex project [12], Prenom-prolex dictionary of first names (more than 6500 entries), acronym-prolex dictionary of abbreviations with their extensions (about 3300 entries) and finally occupation names dictionary [6].

The input alphabet contains patterns to be recognized in texts whereas the output alphabet, in our case, contains in our case information marked out in a language inspired by XML. The patterns we are looking for are proper names and eventually their contexts (if we can locate and exploit them).

Here is an example of a person name found in a text and marked out by the transducer cascade:

Le juge Renaud Van Ruymbeke (the judge Renaud Van Ruymbeke) ⇒

<person><ctxt:profession> juge < \ctxt> <prenom> Renaud < \prenom> <nom> Van Ruymbeke < \nom> < \person>.

The cascade is based on a simple idea: to apply transducers on the text in a precise order to transform or extract patterns from the text. Every pattern discovered is replaced in the text by an indexed label. We eliminate the simplest patterns of the text to avoid that a later transducer extracts them as well.

4.2 A Linguistic Study of Person's Name

Before creating the cascade, we have studied right and left contexts of person names in newspaper articles. Indeed the contexts help to track down proper names. We noticed that the left context allows to discover more than 90 % of the person names in journalistic texts: this is certainly due to stylistic imperatives appropriate to this type of texts which should be objective and should describe facts. A study of an extract from *Le Monde* newspaper (about 165000 words) allowed us to determine the categories of the most frequent contexts.

- Case 1: 25.9 % of person names are preceded by a context containing a title or an occupation name, followed by the first name and by the patronymic name. Ex: *M. Alain Juppé, le président John Kennedy (president John Kenne-dy).*
- Case 2: 19.1 % of person names preceded by a context containing a title or an occupation name followed by a single patronymic, or by an unknown first name (which is not in our dictionary of first names) and finally of a patronymic name. Ex : *le président Chadli.*
- Case 3: 43.4 % of person names with no describable contexts but with a first name (known thanks to our dictionary) and followed by the name of the person. Ex : *Pierre Bourdieu.*
- Case 4: 5.2 % of the forms are located thanks to a verb refering only to human actions (*to say, to explain, etc.*). For example, "*Wieviorka est décédé le 28 décembre*" (*Wieviorka died on December 28*) or "*Jelev a dit...*" (*Jelev said...*). Here we counted appositions too, such as in "*Jospin, premier Ministre ...*" (*Jospin, Prime Minister...*)
- Case 5: The remaining 6.4 % of person names have no context whatsoever that can distinguish them from other proper names. However we noticed that 49 % of the remaining persons' names can yet be detected. Indeed, person names without contexts are mainly very known persons for whom the author considers unnecessary to specify the first name, the title or the profession. It is necessary to realize a second analysis to find the patronymic name, which

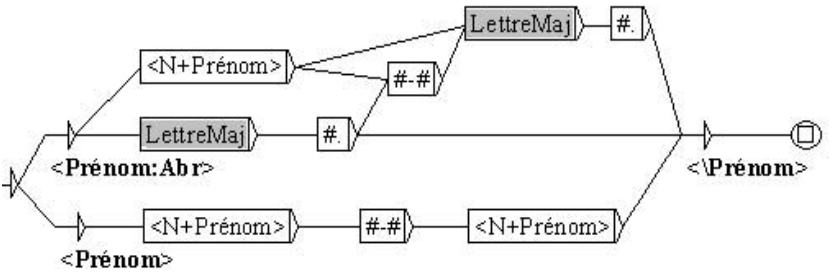


Fig. 3. A transducer describing compound (ex: *John Fitzgerald*) or abbreviated (ex: *J.F.*) first names

one has already discovered in another place of the text; which reduces to 3.3 % the number of undetectable forms. This percentage can still be reduced by a dictionary of celebrity names. Ex : " *Brandissant un portrait de Lénine, ou de Staline, ...*" (*Brandishing a portrait of Lenin, or Stalin, ...*).

4.3 Different Person Name Forms

We also studied different forms of person names. First names followed by a patronymic name or patronymic names alone are most often found. As noticed by [10], the author of a news paper generally gives first the complete form of the person name, then abbreviated forms; that is why the majority of person names are often found with their first name and their last name.

We have described all first name forms (Figure 3[4]) in transducers (using dictionary tags of the text and morphologic clues).

First names unknown to the dictionary are not tagged as first names, but they are included as an integral part of the person's name as *in* <*person*> <*ctxt:titmilit*> *général* < *ctxt*> <*nom*> *Blagoje Adzic* < *nom*> < *Person*> (the person name is *Blagoje Adzic* but we have not distinguished the first name from the patronymic name).

Different patronymic forms are also described using morphology (word beginning with an upper case).

At last contexts are a majority of left contexts which are simply civilities (ex: *Monsieur, Madame, etc.*), titles: politics (ex: *ministre, président , etc.*), nobility titles (ex : *roi (king), duchesse, baron, etc.*), military titles (ex: *général, lieutenant, etc.*), religious titles (ex: *cardinal, Père, etc.*), administration staff (ex: *inspecteur, agent, etc.*) as well as occupation names (ex: *juge, architecte, etc.*). The occupation names are the least frequent terms in contexts. The place-name dictionary allows to track down the adjectives of nationalities in expressions

[4] LettreMaj is an automaton listing upper case letters.

such as "*le président américain Clinton*", "*l'allemand Helmut Kohl*" (*the German Helmut Kohl*).

4.4 Finite-State Cascade Description

According to our various observations on the study of person names and their contexts, we have defined the cascade and given priority to the longest patterns to track down the whole names.

For example, if we apply a transducer that recognizes "*Monsieur*" followed by a word beginning with an upper case letter before the transducer that recognizes "*Monsieur*" followed by a first name (<*prenom*>) then by a name (<*nom*>), and that we have a text containing the sequence "*Monsieur Jean Dupont*", we discover the pattern:

<*person*> <*ctxt:civ*> *Monsieur* <*ctxt*> <*nom*> *Jean* <*nom*> <*person*>

instead of the pattern

<*person*> <*ctxt*> *Monsieur* <*ctxt*> <*prenom*> *Jean* <*prenom*> <*nom*> *Dupont* <*nom*> <*person*>

This is an error because the best parsing is the second.

Then we have designed about thirty transducers to obtain the best results. They are generally constituted of one context part (left or right), a first name part and a patronymic part. But some are only first name and patronymic part, or context and patronymic name. The longest patterns are in the first transducers to be applied.

4.5 Evaluation

Here is an example of results obtained on an article from *Le Monde*. An extract of the original text reads:

Le **président haïtien Aristide** *accepte la candidature de* **M. Théodore** *au poste de premier ministre (...) Avant leur départ pour Caracas, les présidents du Sénat et de la Chambre des députés,* **M. Déjean Bélizaire** *et* **M. Duly Brutus**, *avaient obtenu du " président provisoire " installé par les militaires,* **M. Joseph Nérette**, *l'assurance qu'il démis-sion-ne-rait si les né-go-cia-tions débou-chaient sur la no-mi-na-tion d'un nouveau premier ministre.*{*S*} *(...) Pendant la campagne,* **M. Théodore** *avait concentré ses attaques contre le* **Père Aristide**, *et n'avait cessé de le critiquer après sa triomphale élection.*{*S*}

We finally obtained those extracted patterns:

<*person*> <*ctxt:titpolit*> **président** < *ctxt*> <*ctxt:nation*> **haïtien** < *ctxt*> <*nom*> **Aristide** < *nom*> < *person*>

<*person*> <*ctxt:civ*> **M.** < *ctxt*> <*nom*> **Duly Brutus** < *nom*>< *person*>

<*person*> <*ctxt:civ*> **M.** < *ctxt*> <*nom*> **Déjean Bélizaire** < *nom*> < *person*>

<*person*> <*ctxt:civ*> **M.** <*ctxt*> <*prenom*> **Joseph** <*prenom*> <*nom*> **Nérette** < *nom*> < *person*>

Table 1. Results obtained on an extract of *Le Monde*

	Case 1	*Case 2*	*Case 3*	*Case 4*	*Case 5*	*Total*
Recall	95.7%	99.4%	96.6%	60%	48.7%	91.9%
Precision	98.7%	99.5%	99.2%	94.9%	99.3%	98.9%

<*person*> <*ctxt:civ*> **M.** <*ctxt*><*nom*> **Théodore** <*nom*><*person*>
<*person*> <*ctxt:titeglise*> **Père** < *ctxt*> <*nom*> **Aristide** < *nom*> < *person*>

To verify that the results obtained after this finite-state cascade were correct, we verified a part (about 80000 words) of our corpus of the newspaper *Le Monde*[5] (Table 1). We used the recall and precision measures.

$$Recall = \frac{number\ of\ person\ names\ correctly\ found\ by\ the\ system}{number\ of\ person\ names\ correct\ and\ incorrect\ found\ by\ the\ system}$$

$$Precisionl = \frac{number\ of\ names\ correctly\ found\ by\ the\ system}{number\ of\ person\ names\ really\ present\ in\ the\ text}$$

The results obtained on the first four categories of patterns for person names are very good. we obtained more than 96.9% of recall and more than 99.1 % of precision on the person names preceded by a context and / or by a first name. We notice that in cases 4 and case 5 the results are bad. In case 4, the patterns that surround the person names are very ambiguous ex : "*Microsoft declared*" : the verb *to declare* can be associated to a human being but also to a company as in the example. In case 5, the names are found because they have been found in the text in another context. For example, a text contains the following sentence : "*Ce livre contient des critiques directes au président Mitterrand*" (*This book contains direct criticisms of president Mitterrand*) where the context "*president*" permits to know that *Mitterrand* is a person name. In the same text, we have the sentence "*M. Léotard interpelle Mitterrand sur ...*" (*Mr Léotard calls to Mitterrand on ...*). Thanks to the pattern found before, we know that *Mitterrand* is a person name in this text. Cases 4 and case 5 can be improved during the search of the other names.

5 Conclusion

We present finite-state machines to pre-process a text and locate proper names. The principle of the cascade of transducers is rather simple and effective; on the other hand the description of the patterns to be found turns out to be

[5] Ressources obtained at Elda (`www.elda.fr`)

boring if one wants to obtain the best possible result. Combinations and possible interactions in the cascade are complex. The other proper names (place-names, names of organizations, etc.) are more difficult to track down because their contexts are much more varied.

The results are promising: *Le Monde* is a newspaper of international readership whose journalists respect classic standards and have a concern for precision and details (especially when quoting people and proper names). The results will be worse with other newspapers mainly because of the more approximate style of authors.

Beyond extraction of patterns, bare patterns can serve in numerous domains. One can thus imagine the creation of a system to write XML semi-automatically or to semi-automatically append names to electronic dictionaries.

References

[1] Abney, S. (1996). *Partial parsing via finite-state cascades*, In Workshop on Robust Parsing, 8th European Summer School in Logic, Language and Information, Prague, Czech Republic, pp. 8-15. 116

[2] Ait-Mokhtar, S., Chanod, J. (1997) *Incremental finite state parsing*, in ANLP'97. 116

[3] Coates-Stephens, S. (1993). The Analysis and Acquisition of Proper Names for the Understanding of Free Text, in *Computers and the Humanities*, 26 (5-6), pp. 441-456. 115

[4] Courtois, B., Silberztein, M. (1990). *Dictionnaire électronique des mots simples du français*, Paris, Larousse. 118

[5] Dejong, G. (1982). An Overview of the frump System, in *W. B. Lehnert et M. H. Ringle éd., Strategies for Natural Language Processing*, ErlBaum, pp. 149-176. 115

[6] Fairon, C. (2000). *Structures non-connexes. Grammaire des incises en français : description linguistique et outils informatiques*, Thèse de doctorat en informatique, Université Paris 7. 118

[7] Friburger, N., Dister, A., Maurel, D. (2000). *Améliorer le découpage des phrases sous INTEX*, in Actes des journées Intex 2000, RISSH, Lièges, Belgique, to appear. 116

[8] Gala-Pavia, N. (1999). *Using the Incremental Finite-State Architecture to create a Spanish Shallow Parser*, in Proceedings of XV Congres of SEPLN, Lleida, Spain. 116

[9] Hobbs, J. R., Appelt, D. E., Bear, J., Israel, D., Kameyama, M., Stickel, M., Tyson, M. (1996). FASTUS: A cascaded finite-state transducer for extracting information from natural-language text, in *Finite-State Devices for Natural Language Processing*. MIT Press, Cambridge, MA. 116

[10] Kim, J. S., Evens, M. W. (1996). *Efficient Coreference Resolution for Proper Names in the Wall Street Journal Text*, in online proceedings of MAICS'96, Bloomington. 120

[11] Kokkinakis, D. and Johansson-Kokkinakis, S. (1999). *A Cascaded Finite-State Parser for Syntactic Analysis of Swedish*. In Proceedings of the 9th EACL. Bergen, Norway. 116

[12] Piton, O., Maurel, D. (1997). Le traitement informatique de la géographie politique internationale, in *Colloque Franche-Comté Traitement automatique des*

langues (FRACTAL 97), Besançon, 10-12 décembre, Bulag, numéro spécial, pp. 321-328. 118

[13] Roche, E., Schabes, Y. (1997). *Finite-State Language Processing*, Cambridge, Massachussets, MIT Press. 115

[14] Silberztein, M. (1998). *"INTEX: a Finite-State Transducer toolbox"*, in Proceedings of the 2nd International Workshop on Implementing Automata (WIA'97), Springer Verlag. 116

Is this Finite-State Transducer Sequentiable?

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

Abstract. Sequentiality is a desirable property of finite state transducers: such transducers are optimal for time efficiency. Not all transducers are sequentiable. Sequentialization algorithms of finite state transducers do not recognize whether a transducer is sequentiable or not and simply do not ever halt when it is not. Choffrut proved that sequentiality of finite state transducers is decidable. Béal et al. have proposed *squaring* to decide sequentiality. We propose a different procedure, which, with ε-closure extension, is able to handle letter transducers with arbitrary ε-ambiguities, too. Our algorithm is more economical than squaring, in terms of size. In different cases of non-sequentiability necessary and sufficient conditions of the ambiguity class of the transducer can be observed. These ambiguities can be mapped bijectively to particular basic patterns in the structure of the transducer. These patterns can be recognized, using finite state methods, in any transducer.

1 Introduction

Finite-state automata and transducers are widely used in several application fields, among others, in computational linguistics. **Sequential** transducers, introduced by Schützenberger [13], have advantageous computational properties. Sequentiality means determinism on the input side (automaton) of the underlying relation the transducer encodes.

We use the notation of the Xerox finite-state calculus [6, 7, 8]. In particular, the *identity relation* `a:a` will be referred to as `a` and the *unknown symbol* as `?`. The main application area of the Xerox calculus is natural language processing. As usual, the word "sequential" will be used as a synonym for subsequential and p-subsequential unless a distinction is needed. "Letter transducer" means a format where arcs have pairs of single symbols on arcs, in "word" format they have pairs of possibly several symbols – words. Even if one format can be transformed into the other, the distinction is necessary in practical applications, like natural language processing: among other considerations, since there tend to be more words than letters in a human language, much better compaction can be achieved by using the letter form.

While any finite-state automaton can be determinized, not all transducers can be sequentialized. Choffrut proved [2] that sequentiability of finite state transducers is decidable: the proof is based on a distance of possibly ambiguous paths.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 125–134, 2002.
© Springer-Verlag Berlin Heidelberg 2002

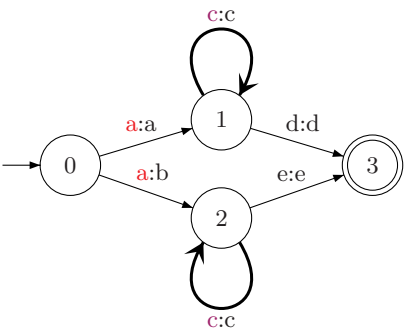


Fig. 1. A non-sequentiable transducer: there is no sequential equivalent since arbitrarily big possible delay of output cannot be handled by sequentialization: an input string starting with $\texttt{ac}^n$ can either give $\texttt{ac}^n\texttt{d}$ or $\texttt{bc}^n\texttt{e}$ output depending on the last input symbol only so the decision must be delayed until this last symbol arrives and this can go beyond any predefined bound. Note that the transduction is functional but not sequential

Mohri [11] gave a generalization of the theorem of Choffrut for p-subsequential transducers.

It has been known in finite-state folklore that sequentiability can be decided by using the *square* construct. Roche and Schabes [12] described two algorithms to decide the sequentiality of *unambiguous* transducers; one of them is the twinning construct of Choffrut.

Béal et al. have published a formal paper [1] on squaring transducers where they describe the proof and give algorithms, using the square, to decide *functionality* and *sequentiability* of transducers. The algorithm we propose decides about sequentiability only.

Our method has the advantage of not having to create the square of the transducer. If a transducer has n states, its square, as one would expect, will have n^2 ones. Automata implementations, like the Xerox automata [8, 6], often have practical limits in terms of states and arcs. Even if these limits are pushed further, and even if properties of particular implementations are ignored, the size complexity remains both a concern and a real limit.

In [3] we published extensions to the sequentialization algorithm of Mohri [9, 10]. One of them was the handling of not only real, but ε-ambiguities, too. It is necessary when the transducer is in letter format since then one-sided ε-transitions may not be eliminated, while it is possible in the word format.

To determine the sequentiability of letter transducers ε-ambiguities have to be handled too, unless we can guarantee ε-free letter transducers. But in the general case there is no such guarantee. Handling ε–ambiguities needed some improvements in our original algorithm to decide about sequentiability.

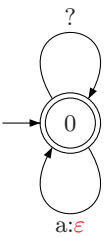


Fig. 2. The `[ a -> ε ]` replace expression causes an ε-loop in the corresponding transducer. It is infinitely ambiguous from the lower side (but not from the upper side). The relation eliminates all `a`s from the upper side language

Transducers can be building blocks for more complicated transducers, both in finite state compilers (like the Xerox [8] one) and in other applications. Computational linguists often build transducers that can serve for both generation and analysis of text. Such transducers can have various levels of ambiguities, and the level of ambiguity characterizes the given (input) side. Roche and Schabes classify the level of ambiguity of transductions into four classes ([12], 1.3.5) and in simple applications of some basic constructs like the replace operator [5], the least convenient, that is, the most general case, an *infinitely ambiguous* transducer can easily be created, as in Fig. 2.

In the following, transducers will be considered as those with accessible states, in a connected graph, only. It is a practical consideration since this case corresponds to the language of regular expressions.

2 What Makes a Transducer Non-sequentiable

If a transducer represents a *sequential mapping* it can be sequentialized. An example is in Fig. 7 which does represent a sequential mapping but is not sequential; note that here the ε-closure extension is needed for sequentialization. The sequentialization algorithm[1] attempts to find ambiguities and possibly delay the output on ambiguous paths until non-determinism can be resolved. In a finite transducer, this can go to a finite distance only. In the terminology of Choffrut, the transducer must have *bounded variation* (Béal et al. [1] call this property *uniformly divergent*).

If a transducer contains an ε-loop then it is infinitely ambiguous. Such a transducer does not represent a sequential mapping, examples are in Fig. 2 and 3.

An intuitive interpretation of this case is that an infinitely ambiguous transducer, considered from the given input direction, can give several, possibly infinitely many, different results for an input string. In the example of Fig. 3,

[1] Both that of Mohri [9] and our variant [3].

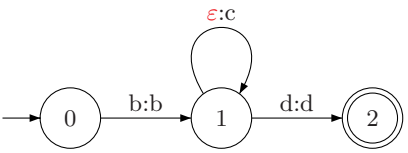


Fig. 3. The transducer of the `[ b ε:c* d]` expression is infinitely ambiguous from the upper side, yielding an infinity of results at lookup

looking from the *upper* direction, at an input `bd` string, the result is the infinite set $\texttt{bc}^n\texttt{d}$ (where n can be any natural number). In fact, in all transducers, having this ambiguity property, one can find an ε-loop. So if the presence of ε-loops can be detected, this condition, which excludes sequentialization, can be found.

If a transducer is *unboundedly ambiguous* (Roche and Schabes call this *simply finitely ambiguous*), it is not sequentiable either.

Intuitively, such a transducer gives an ever growing number of different results if the length of the input is growing. There is no upper limit for the number of results. Such a transducer does not have *bounded variation*. In the example of Fig. 4 the number of results is 2^n where n is the number of input `a`s and 2 characterizes the output variation (`b` or `c`), since we may have `b` or `c` at each output position.

The same example can be made somewhat more obfuscated to show the effect of ε-ambiguities: Fig. 5 is the same mapping as in Fig. 4, just in a nastier representation, so it is also unboundedly ambiguous. In addition, the number of results is not only much bigger, but it grows much faster at greater n, too, than in the previous example. Since there are three loops that encode the same mapping, the number of results for n input `a`s is 2^n3^n, of which 2^n different ones. 2 characterizes the output variation, as before, and 3 is the number of ambiguous paths (for each new input symbol).

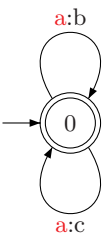


Fig. 4. Unboundedly ambiguous transducer, `[ a:b | a:c ]*` , from the upper side

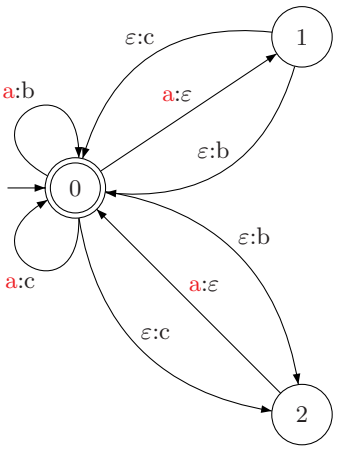


Fig. 5. Unboundedly ambiguous transducer spiced with ε-ambiguities: it represents the same mapping as Fig. 4, but looks more complicated, and, for pattern matching, it is, indeed

In the first unboundedly ambiguous example (Fig. 4) the pattern to detect in the transducer is the following: if there is ambiguity on a state, and if the ambiguous arcs have different outputs, and if these paths get to (possibly different) loops with the same input mapping, then such a transducer is not sequentiable, since it is unboundedly ambiguous (at least). The second example (Fig. 5) shows that even this simple mapping might not be that easy to detect: in [3] we showed that many different paths can encode the same relation in transducers with one-sided ε-ambiguities. The number of possible identical paths (involving one-sided ε-ambiguities) grows very fast with the increase of the length of the relation. For this reason, this condition may not be obvious to identify in complicated structures of large transducers – but, with some effort, it can be done. This effort is the ε-closure of states so that we know all the ambiguities of a particular state, let they be directly on the state or at arbitrary distance (throughout one-sided ε-arcs). The creation of the ε-closure set is known [3].

By now we know everything to detect sequentiability – or almost. Any other transducer, not falling into the previous two ambiguity classes, represents a sequential mapping, and is sequentiable. They do not exceed the *uniformly finitely ambiguous class* of transducers. We have to look only for the two excluding conditions above, that is, first for ε-loops and then for loops that begin ambiguously, when testing transducers for sequentiability.

As a direct consequence of the above, any non-cyclic transducer is sequentiable since such a transducer does not contain any loop.

The rest will explain in more detail how to detect such patterns, forbidding sequentialization, in transducers, using reasonably simple algorithms and finite-state methods.

3 Exclude Infinitely Ambiguous Transducers

A transducer that contains an ε-loop is infinitely ambiguous, see Roche and Schabes [12], 1.3.5. Such a transducer is not sequential, and cannot be sequentialized. We have seen before that such a situation can easily arise in commonly used transducers. It is a trivial case of non-sequentiability, and it is quite simple to detect the presence of epsilon loops in a network. A recursive exploration of possible (input-) ε-arcs on all the states can do it, as in Fig. 6.

This algorithm is to be performed first; and only those transducers that have passed this test should undergo more thorough scrutiny. The reason is that the test to detect the presence of *unbounded ambiguity* is not able to detect the presence of ε-loops, worse, it either would not halt if such a pattern occurred in the transducer or would not recognize it as an obstacle in sequentiability.

4 Exclude Unboundedly Ambiguous Transducers

In Section 2 we have introduced unboundedly ambiguous transducers and identified a pattern which is always present in a transducer having this ambiguity property.

```
   HAS_EPSILON_LOOP(T)
 1 for all states s in T
 2     if STATE_EPSILON_LOOP(s,s)
 3         return TRUE
 4 return FALSE

   STATE_EPSILON_LOOP(s0, s1)
 1 if s1 has been VISITED
 2     return TRUE
 3 Mark s1 as VISITED
 4 for all arcs a in s1
 5     if input symbol of a is ε and a has not been VISITED
 6         Mark a as VISITED
 7         if Destination_state(a) = s0
 8             return TRUE
 9         else
10             if STATE_EPSILON_LOOP(s0, (Destination_state(a))
11                 return TRUE
12     Mark a as NOT-VISITED
13 Mark s1 as NOT-VISITED
14 return FALSE
```

Fig. 6. Algorithm to discover ε-loops in a transducer. If there is an ε-loop then the transducer is infinitely ambiguous hence non-sequentiable

If there is no (real- or ε-) ambiguity on any state, there is no need to check for unbounded ambiguity: such a transducer can still be infinitely ambiguous (as in Figures 2 and 3) so we have to exclude this by testing this first, as in Fig. 6.

Testing for unbounded ambiguity is done only when necessary, that is, when there is ambiguity, at the first place. Ambiguity can be due to ambiguous arcs on the state itself (as in Fig. 4) or due to a mixture of real and ε-ambiguities (as in Fig. 5), or just due to pure ε-ambiguities. Any ambiguity must have a beginning, that is, a particular state where there are ambiguous arcs – either own arcs of the state or arcs in the ε-closure of the state. An iteration on the set of all states of the transducer, using ε-closure, is able to identify ambiguities.

If a state has (at least) two ambiguous arcs, they define, via the closures of their respective destination states, two sub-transducers. If both of these are looping then there is further work to be done, otherwise the current arc pair cannot lead to unbounded ambiguity.

If they are both looping but the input substrings, which loop at least once, are different, then there is no problem. But if it is not the case we may have found unbounded ambiguity, so, in most cases, the test could stop here and report non-sequentiability. It is the case in Fig. 1.

But there is still a little chance that such a transducer is sequentiable, notably when the current two sub-transducers represent the same mapping but it is hidden by ε-ambiguities. It is only possible in transducers where there can be ε-ambiguities, as in Fig. 7.

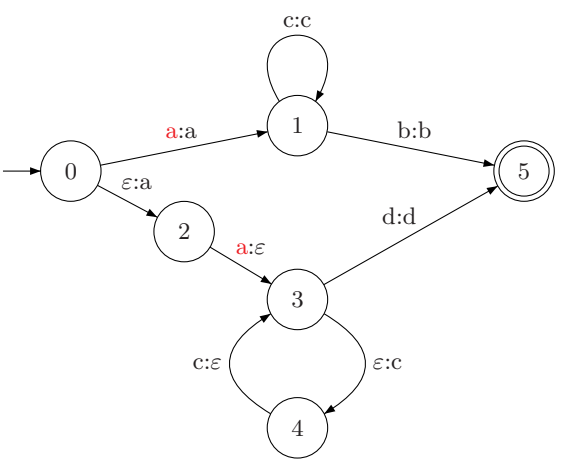


Fig. 7. A sequentiable transducer: since there are ambiguous arcs that lead to loops, the test has to examine if there is real unbounded ambiguity or identity. In this case, the ambiguous sub-transducers, with loops, hide identity

Both the condition of unbounded ambiguity and the eventual hidden identical mapping can be found by examining respective sides of the sub-transducers.

One has to extract sub-transducers: it can be done by considering the current state as initial state, with the current two ambiguous arcs as the respective single arc of this state and traversing this transducer (in a concrete implementation, copying or marking it, too; for both arcs).

The condition of looping can be examined by systematic traversals of the extracted subnets. If, starting from a state, and traversing the net, the current state is reached again then this state is part of a loop and so the whole transducer is looping. This has to be done for all states (of the respective subnets).

If both subnets are looping, then one has to create the *intersection* of their input languages. If this automaton is also looping, then it means that an ambiguous path gets into a loop. It may well mean unbounded ambiguity. The only escape is when respective output languages of the two sub-transducers are identical, too. One has to check the intersection of the output sides of the current sub-transducers and if the intersection of them is not equivalent with its inputs then it is indeed a case of unbounded ambiguity, and so the transducer is not sequentiable.

Fig. 8 shows it more concisely: *Closured_input_symbol()* of line 2 means the possible ε-closure of an arc. *Extract_transducer()* (lines 3 and 4) has been explained earlier. *Input_automaton()*, respectively *Output_automaton ()* (lines 5, 6 and 10, 11) represent the appropriate (upper or lower) sides of the transducer,

```
   HAS_UNBOUNDED_LOOPS_WITH_NON_IDENTICAL_OUTPUT(T)
 1 for all states s in T
 2         for all arcs a_i, a_j in s so that Closured_input_symbol(
   a_i)=Closured_input_symbol( a_j)
 3       T_i = Extract_transducer ( a_i)
 4       T_j = Extract_transducer ( a_j)
 5       A_i^input = Input_automaton ( T_i)
 6       A_j^input = Input_automaton ( T_j)
 7       if Has_loop( A_{i_input_i}) and Has_loop( A_j^input)
 8           A_ij^input = Intersect(a_i, a_j)
 9           if Has_loop( A_ij^input)
10               A_i^output = Output_automaton ( T_i)
11               A_j^output = Output_automaton ( T_j)
12               if A_i^output ≠ A_j^output
13                   return TRUE
14 return FALSE
```

Fig. 8. Algorithm to discover ambiguous loops with identical input substrings that start ambiguously, and then loop. If such loops are found, and they do not hide identical mappings (via ε-ambiguities) then the transducer is unboundedly ambiguous hence non-sequentiable

that is, the corresponding automata. *Has_loop()* (lines 7, 9) is a known basic algorithm, it decides whether the automata is loop-free or not. *Intersect()* of line 8 is intersection (of two automata) in the automata sense. The *equivalence* of two automata (line 12) is decidable.

4.1 Epsilon-Closure of Transducers Representing a Sequential Mapping

In [3] we describe the necessity of allowing ε-transitions in the case of letter transducers. For this reason we could not use the sequentiability tests described by Roche and Schabes since they needed unambiguous transducers as input for their algorithms. In the case of letter transducers we cannot guarantee this.

5 Summary

We have described an algorithm to decide whether a transducer represents a p-subsequential mapping or not. The algorithm is the orderly application of the algorithms in Fig. 6 and 8, in this order. The transducer can be either in letter or in word format and it can contain ε-ambiguities.

As a fringe benefit, the algorithm is able to decide whether a transducer, representing a p-subsequential mapping, is already sequential (no ambiguous arcs then) or not.

The algorithm minimizes unnecessary work: it only explores further paths when needed, that is, when there is possibility of subsequentiability due to real or ε-ambiguities at a given state.

Based on the classification of possible ambiguities, and corresponding patterns in the transducers, these patterns are recognized by examining appropriate input (and, in some cases, the output) sub-languages of the transducer.

If an ε-ambiguous, letter transducer indeed represents a p-subsequential relation then it may or may not be already p-subsequential. If it is not, it can be converted to an optimal p-subsequential transducer by another algorithm, shortly outlined at CIAA2000 ([4]), detailed in [3]. This latter algorithm is based on previous work of Mohri. The test of sequentiality is necessary for all practical purposes – as in finite state compilers and applications – since, applied to arbitrary transducers, the sequentialization algorithm may not halt.

We have implemented these algorithms in the Xerox finite state toolkit.

Acknowledgement

I would like to express my gratitude for the never ceasing helpful attention of Lauri Karttunen. He and Ron Kaplan provided valuable help and context. The figures were created by the GASTeX package of Paul Gastin, transcripted from figures created by the VCG tool of Georg Sanders. The original examples were created by the Xerox finite state tools.

References

[1] Marie-Pierre Béal, Olivier Carton, Christophe Prieur, and Jacques Sakharovitch. Squaring transducers: An efficient procedure for deciding functionality and sequentiality. In D. Gonnet, G. Panario and A. Viola, editors, *Proceedings of LATIN 2000*, volume 1776, pages 397–406. Springer, Heidelberg, 2000. LNCS 1776. 126, 127

[2] Christian Choffrut. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationelles. *Theoretical Computer Science*, 5(1):325–337, 1977. 125

[3] Tamás Gaál. Extended sequentializaton of finite-state transducers. In *Proceedings of the 9th International Conference on Automata and Formal Languages (AFL'99)*, 1999. Publicationes Mathematicae, Supplement 60 (2002). 126, 127, 129, 133

[4] Tamás Gaál. Extended sequentializaton of transducers. In Sheng Yu and Andrei Pâun, editors, *Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA 2000)*, pages 333–334, Heidelberg, 2000. Springer. LNCS 2088. 133

[5] Lauri Karttunen. The replace operator. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics. ACL-95*, pages 16–24, Boston, Massachusetts, 1995. ACL. 127

[6] Lauri Karttunen and Kenneth R. Beesley. *Finite-State Morphology: Xerox Tools and Techniques*. Cambridge University Press, Cambridge UK, 2002? Forthcoming. 125, 126

[7] Lauri Karttunen, Jean-Pierre. Chanod, Gregory Grefenstette, and Anne Schiller. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328, 1996. CUP Journals (URL: www.journals.cup.org). 125

[8] Lauri Karttunen, Tamás Gaál, Ronald M. Kaplan, André Kempe, Pasi Tapanainen, and Todd Yampol. Finite-state home page. `http://www.xrce.xerox.com/competencies/content-analysis/fst/`, Xerox Research Centre Europe, 1996-2002. Grenoble, France. 125, 126, 127

[9] Mehryar Mohri. Compact representation by finite-state transducers. In *Proceedings of the 32nd meeting of the Association for Computational Linguistics (ACL 94)*, 1994. 126, 127

[10] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, pages 269–312, 1997. 126

[11] Mehryar Mohri. On the use of sequential transducers in natural language processing. In *Finite-State Language Processing*, chapter 12, pages 355–378. MIT Press, Cambridge, Massachusetts, USA, 1997. 126

[12] Emmanuel Roche and Yves Schabes, editors. *Finite-State Language Processing*. MIT Press, Cambridge, Massachusetts, USA, 1997. 126, 127, 130

[13] Marcel-Paul Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, 1977. 125

Compilation Methods of Minimal Acyclic Finite-State Automata for Large Dictionaries*

Jorge Graña, Fco. Mario Barcala, and Miguel A. Alonso

Departamento de Computación, Facultad de Informática, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
{grana,barcala,alonso}@dc.fi.udc.es

Abstract. We present a reflection on the evolution of the different methods for constructing minimal deterministic acyclic finite-state automata from a finite set of words. We outline the most important methods, including the traditional ones (which consist of the combination of two phases: insertion of words and minimization of the partial automaton) and the incremental algorithms (which add new words one by one and minimize the resulting automaton on-the-fly, being much faster and having significantly lower memory requirements). We analyze their main features in order to provide some improvements for incremental constructions, and a general architecture that is needed to implement large dictionaries in natural language processing (NLP) applications.

1 Introduction

Many applications of NLP, such as tagging or parsing a given sentence, can be too complex if we directly deal with the stream of input characters forming the sentence. Usually, a previous step of processing changes those characters into a stream of higher level items (called *tokens* and that typically are the words in the sentence), and obtains the candidate tags for these words rapidly and comfortably. This previous step is called lexical analysis or *scanning*.

The use of finite-state automata to implement efficient scanners is a well-established technique [1]. The main reasons for compressing a very large dictionary of words into a finite-state automaton are that its representation of the set of words is compact, and that looking up a word in the dictionary is very fast (proportional to the length of the word) [4]. Of particular interest for NLP are minimal acyclic finite-state automata, which recognize finite sets of words.

This kind of automata can be constructed in various ways [7]. This paper outlines the most important methods and analyzes their main features in order to propose some improvements for the algorithms of incremental construction. The motivation of this work is to build a general architecture to handle suitably two large Spanish dictionaries: the GALENA lexicon (291,604 words with 354,007

* This work has been partially supported by the European Union (under FEDER project 1FD97-0047-C04-02), by the Spanish Government (under project TIC2000-0370-C02-01), and by the Galician Government (under project PGIDT99XI10502B).

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 135–148, 2002.
© Springer-Verlag Berlin Heidelberg 2002

possible taggings) and the ERIAL lexicon (775,621 words with 993,703 possible taggings)[1].

Section 2 describes our general model of dictionary and allows us to understand the role of the finite-state automata here. In Sect. 3, we give the formal definitions and explain how to establish a perfect hashing between the words and their positions in the dictionary, simply by assigning a *weight* to each state [5]. Section 4 recalls a minimization algorithm owing to Revuz [6], which is based on another property of the states: the *height*. Section 5 recalls the incremental construction by Daciuk [2], which performs insertions and minimizations at the same time, by storing in a *register* the states that will conform the final automaton. In Sect. 6, we combine *weights* and *heights* to improve the accesses to the *register*, and compare our implementation with the previous ones. Section 7 presents the conclusion after analysing the data obtained.

2 Compact Modeling of a Dictionary

Many words in a dictionary are manually inserted by linguists to exhaustively cover the invariant kernel of a language (articles, prepositions, conjunctions, etc.) or the terminology of a specific field. But many other words can be captured from annotated texts, making possible to obtain additional information, such as the frequency of the word or the probability with respect to each of its possible tags. This information is essential in some applications, e.g. stochastic tagging and parsing.

Therefore, our first view of a dictionary is simply a text file, with the following line format: `word tag lemma probability`. Ambiguous words use one different line for each possible tag. With no loss of generality, the words could be alphabetically ordered. Then, in the case of the `Galena` lexicon, the point in which the ambiguity of the word `sobre` appears could have this aspect[2]:

```
sobre P sobre 0.113229
sobre Scms sobre 0.00126295
sobre Vysps0 sobrar 0.0117647
```

For a later discussion, we say that the GALENA lexicon has $M = 291,604$ different words, with $L = 354,007$ possible taggings. This last number is precisely the number of lines in the text file. The first tagging of `sobre` appears in the line $325,611$, but the word takes the position $268,249$ in the set of the M different lexicographically ordered words.

Of course, this is not an operative version for a dictionary. Therefore, what is important now is to provide a compiled version to compact this great amount

[1] GALENA is *Generation of Natural Language Analyzers* and ERIAL is *Information Retrieval and Extraction Applying Linguistic Knowledge*. See `http://coleweb.dc.fi.udc.es` for more information of both projects.

[2] The tags come from the GALENA tag set, which has a cardinal of $T = 373$ tags. The meanings for the tags (and for the word `sobre`) is the following: `P` is preposition (on); `Scms` is substantive, common, masculine, singular (envelope); and `Vysps0` is verb, first or third person, singular, present, subjunctive (to remain, to be superfluous).

of data, and also to guarantee an efficient access to it with the help of automata. The compiled version is shown in Fig. 1, and its main elements are:

- The `Word_to_Index` function (explained later) changes a word into its relative position in the lexicon (e.g. `sobre` into 268, 249).
- In a *mapping* array of size $M + 1$, this number is changed into the absolute position of the word (e.g. 268, 249 into 325, 611).
- This new number is used to access the arrays of *tags*, *lemmas* and *probabilities*, all of them of size L.
- The array of *tags* stores numbers, which are more compact than the names of the tags. Those names can be recover from the *tag set* array, of size T. The lexicographical ordering guarantees that the tags of a given word are adjacent, but we need to know how many they are. For this, it is enough to subtract the absolute position of the word from the value of the next cell (e.g. $325,614 - 325,611 = 3$ tags). This is also valid to correctly access the arrays of *lemmas* and *probabilities*.
- The array of *lemmas* also stores numbers. A lemma is a word that also has to be in the lexicon. The number obtained by the `Word_to_Index` function for this word is the number stored here, since it is more compact than the lemma itself. The original lemma can be recovered by the `Index_to_Word` function (explained later).
- The array of *probabilities* directly stores the probabilities. In this case, no reduction is possible.

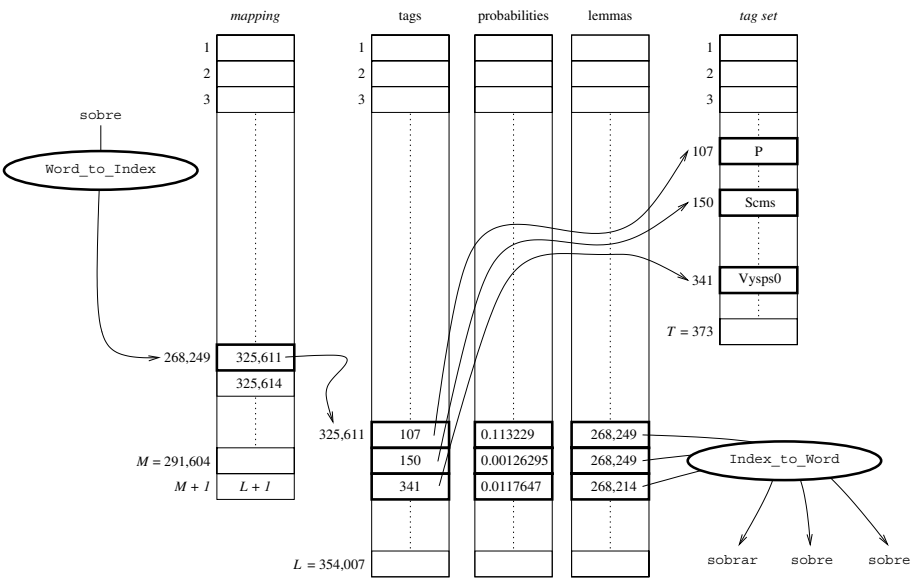


Fig. 1. Compact modeling of a dictionary

This is the most compact architecture for storing all the lexical information of the words present in a dictionary, when this information involves specific features of each word, such as the probability. Furthermore, this architecture is very flexible: it is easy to incorporate new arrays for other additional data (such as frequencies), or to remove the non-used ones (saving the corresponding space).

To complete this model, we only need the implementation of `Word_to_Index` and `Index_to_Word`. Both functions operate over a special type of automata, the numbered minimal acyclic finite-state automata described in the next section.

3 Numbered Minimal Acyclic Finite-State Automata

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

- Q is a finite set of states (the vertices of the underlying graph),
- Σ is a finite alphabet of the input symbols that conform the words (the labels in the transitions of the graph),
- δ is a function of $Q \times \Sigma$ into 2^Q defining the transitions of the automaton,
- q_0 is the initial state (the entrance of the graph), and
- F is the subset of final states of Q (usually marked with thicker circles).

The state or set of states reached by the transition of label a of the state q is denoted by $q.a = \delta(q, a)$. When this is only one state, i.e. when δ is a function of $Q \times \Sigma$ into Q, the automaton is *deterministic*. The notation is transitive: if w is a word then $q.w$ denotes the state reached by using the transitions labelled by each letter w_1, w_2, ..., w_n of w. A word w is accepted by the automaton if $q_0.w$ is in F. We define $\mathcal{L}(A)$, the *language recognized* by an automaton A, as the set of words w such that $q_0.w \in F$. An *acyclic* automaton is one such that the underlying graph is acyclic.

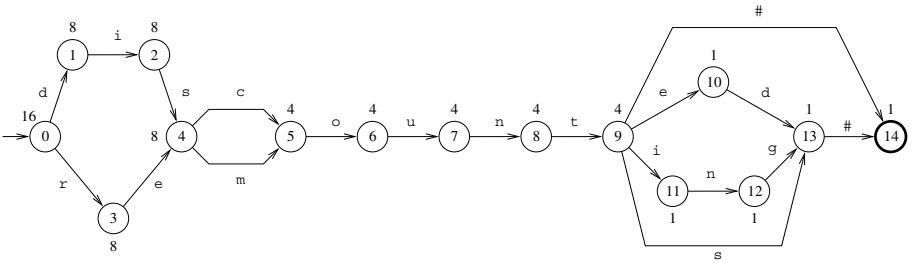
Deterministic acyclic automata are the most compact structure for recognizing finite sets of words. The ratios of compression are excellent and the recognition times are linear with respect to the length of the word to be scanned. The remaining sections of this paper will present several methods to obtain the minimal deterministic acyclic automaton for any finite set of words.

However, this is not enough for our model of dictionaries. We need a mechanism to transform every word into a univocal numeric key and viceversa. This transformation can easily be done if the automaton incorporates a *weight* for each state, this weight being the cardinal of the *right language* of the state, i.e. the number of substrings accepted from this state [5]. We refer to these automata as *numbered minimal deterministic acyclic finite-state automata*.

Figure 2 shows the numbered minimal automaton that recognizes all the forms of the English verbs `discount`, `dismount`, `recount` and `remount`[3]. The assignment of the indexing weights can be done by a simple recursive traversal of the automaton, when it has been correctly built and minimized.

Now, we can give the details of the functions that perform the hashing between the words in the lexicon and the numbers 1 to M (the size of the lexicon).

[3] The symbol `#` denotes the end of string.


16
8
8
8
4
4
4
4
4
4
1
1
1

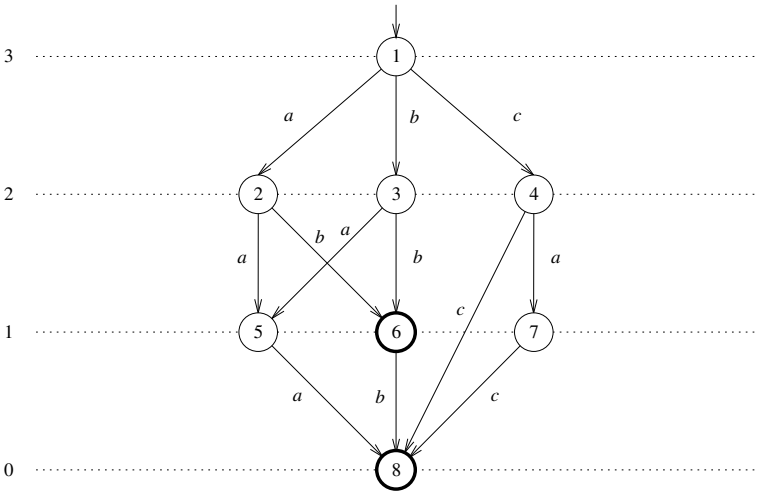


Fig. 3. A non-minimal acyclic deterministic finite-state automaton

same language as the original one [4]. Furthermore, if the automaton is acyclic, this process is simpler, as we will see through the rest of the paper.

On the other hand, and also due to the same memory requirements, it is not convenient to build a dictionary by inserting all the words in a tree and then obtaining the minimal automaton corresponding to that tree. Instead of this, it is more advisable to perform several steps of insertion and minimization[5].

In any case, to formally define the base of traditional minimization algorithms [6], we need the following definitions. Two automata are *equivalent* if they recognize the same language. Two states p and q are *equivalent* if the subautomaton with p as initial state and the one that starts in q are equivalent. The opposite concept is that two states are non-equivalent or *distinguished*. If A is an automaton, there exists a unique automaton $\mathcal{M}$ minimal by the number of states, recognizing the same language, i.e. $\mathcal{L}(A) = \mathcal{L}(\mathcal{M})$. An automaton with no pair of equivalent states is minimal. Now, for a state s, we define its *height* $h(s) = max\{|w| \mid s.w \in F\}$, i.e. the height of a state s is the length of the longest path starting at s and leading to a final state. This function gives a partition Π of Q. Π_i denotes the set of states of height i. We say that the set Π_i is *distinguished* if no pair of states in Π_i is equivalent.

In Fig. 3 we show an automaton recognizing the language $\mathcal{L} = \{aaa, ab, abb, baa, bb, bbb, cac, cc\}$. States of the same height are drawn on the same dotted

[5] In [3] we describe how words must be properly inserted into an already minimized partial automaton in order to avoid inconsistencies. The basic idea is to clone conflicting states that can give rise to unintentional insertions of words not present in the original lexicon. Furthermore, we also give an empirical reasoning of the maximum size of the automaton needed to obtain a reasonable balance between the number of insertion-minimization steps and their duration.

line. This automaton is not minimal. States 2 and 3 of height 2 are equivalent. We can collapse these states by removing one of them, e.g. state 2, and replacing the target of its entering transitions by the other state, i.e. $1 \xrightarrow{a} 2$ by $1 \xrightarrow{a} 3$.

Now we can state the *height property*: if every Π_j with $j < i$ is distinguished, then two states p and q in Π_i are equivalent if and only if for any letter a in Σ the equality $p.a = q.a$ holds. The minimization algorithm by Revuz [6], in Fig. 6 of appendix A, follows from the height property. First we create a partition by height which is calculated by a standard traversal of the automaton for which the time complexity is $\mathcal{O}(t)$, where t is the number of transitions. If the automaton is not a tree, some speedup can be realized with a flag showing that the height of a state is already computed, and useless states which have no height can be eliminated during the traversal. Then, every Π_i is processed, from $i = 0$ to the height of the initial state, by sorting the states according to their transitions and collapsing equivalent states.

Using a sorting scheme with a time complexity $\mathcal{O}(f(e))$, where e is the number of elements to sort, the algorithm of Fig. 6 minimizes an acyclic automaton in

$$\mathcal{O}(t + \sum_{i=0}^{h(q_0)} f(|\Pi_i|))$$

which is less than the minimization algorithm by Hopcroft for general finite-state automata: $\mathcal{O}(n \times \log n)$, where n is the number of states [4].

This process needed 10 steps of insertion-minimization to build the minimal acyclic automaton for the Galena lexicon (11,985 states and 31,258 transitions), and took 29 seconds in a Pentium II 300 MHz. under Linux operating system.

5 Algorithms for Incremental Construction

As we have seen, traditional methods for constructing minimal acyclic automata from a finite set of words consist of two phases: the first being to construct a tree or a partial automaton, the second one being to minimize it. However, there are methods of incremental construction able to perform minimization in-line, i.e. at the same time as the words are inserted in the automaton [2]. These methods are much faster and have significantly lower memory requirements.

To build the automaton one word at a time, we need to merge the process of adding new words with the minimization process. There are two crucial questions that must be answered:

1. Which states are subject to change when new words are added?
2. Is there a way to add new words such that we minimize the number of states that may need to be changed during the addition of a word?

If the input data is lexicographically ordered, only the states that need to be traversed to accept the previous word added to the automaton may change when a new word is added. The rest of the automaton remains unchanged, because a new word either:

- begins with a symbol different from the first symbols of all words already in the automaton (in this case, the beginning symbol of the new word is lexicographically placed after those symbols); or
- it shares some initial symbols of the word previously added (in this case, the algorithm locates the last state in the path of the common prefix and creates a forward branch from that state, since the symbol on the label of the new transition must be later in the alphabet than symbols on all other transitions leaving that state).

Therefore, when the previous word is a prefix of the new word, the only states that can change are the states in the path of the previous word that are not in the path of the common prefix. The new word may share its ending with other words already inserted, which means that we need to create links to some parts of the automaton. Those parts, however, are not modified.

Now we describe the algorithm of incremental construction from a finite set of words in the lexicographical order. This algorithm, which is shown in Figs. 7 and 8 of appendix A, uses a structure called *Register* that always keeps a representative state of every equivalence class of states in the automaton. Therefore, the *Register* is itself the minimal automaton in every step.

The main loop of the algorithm reads subsequent words and establishes which part of the word is already in the automaton (the *Common_Prefix*), and which is not (the *Current_Suffix*). An important step is determining what the last state in the path of the common prefix is (the *Last_State*). If *Last_State* already has children, it means that not all states in the path of the previously added word are in the path of the common prefix. In that case, by calling the function `Replace_or_Register`, we let the minimization process work on those states in the path of the previously added word that are not in the common prefix path. Then we add to the *Last_State* a chain of states that recognize the *Current_Suffix*.

The function `Replace_or_Register` effectively works on the last child of the argument state. It is called with the argument that is the last state in the common prefix path (or the initial state in the last call). We need the argument state to modify its transition in those instances in which the child is to be replaced with another equivalent state that has already been registered. Firstly, the function calls itself recursively until it reaches the end of the path of the previously added word. Note that when it encounters a state with more than one child, it always takes the last one. As the length of words is limited, so is the depth of recursion. Then, returning from each recursive call, it checks whether a state equivalent to the current state can be found in the register. If this is true, then the state is replaced with the equivalent state found in the register. If not, the state is registered as a representative of a new class. Note that this function processes only those states belonging to the path of the previously added word, and that those states are never reprocessed.

In the same paper [2], the authors also propose an incremental construction method for unsorted sets of words, which is also based on the clonation of states that become conflicting as new words are added. The method is slower and uses

more memory, but it is suitable when the sorting of the input data is complex and time-consuming.

6 Improving the Access to the Register

During the incremental construction, the automaton states are either in the register or on the path for the last added word. All the states in the register are states in the resulting minimal automaton. Hence the temporary automaton built during the construction has fewer states than the resulting automaton plus the length of the longest word. As a result of this, the space complexity is $\mathcal{O}(n)$, i.e. the amount of memory needed by the algorithm is proportional to n, the number of states in the minimal automaton. This is an important advantage of the algorithm.

With regard to the execution time, the algorithm presents two critical points which are marked with boxes in Fig. 8 of appendix A. This means that the time complexity will depend on the data structure implemented to perform the searches of equivalent states and the insertions of new representative states in the register. In [2], the authors suggest that, by using a hash table to implement the register and its equivalence relations, the time complexity of those operations can be made almost constant and equal to $\mathcal{O}(\log n)$.

Unfortunately, such a hashing structure is not described, although it can be deduced directly from the C++ implementation of the algorithm made freely available by the authors at `http://www.pg.gda.pl/~jandac/fsa.html`. This implementation took 3.4 seconds to build the minimal acyclic automaton for the Galena lexicon (11,985 states and 31,258 transitions) and 11.2 seconds to build the one for the Erial lexicon (52,861 states and 159,780 transitions), in a Pentium II 300 MHz. under Linux operating system.

Here, instead of a detailed study of that code, we prefer to detail our own implementation, since we think it automatically integrates some features that are needed in the general architecture of dictionaries presented in Sect. 2, and we have checked that is faster, as we will see later.

When a given state is subject to be replaced or registered, it must be compared with the states already present in the register. Of course, we cannot compare it with all these states, because the register becomes greater and greater as we insert new words in the automaton. Then, we have to think again: When are two states equivalent? We find the following answers for this question, each of them constituting a new filter that leaves more and more states out of the comparison process:

- Given two states, their heights have to be equal if the states are to be equivalent. The height is not specifically needed either for the incremental algorithm or for the dictionary scheme, but it nevertheless constitutes an effective filter.
  Furthermore, the height is a relatively low number (ranging between 0 and the length of the longest word), and it can be calculated in-line with no extra

traversal of the automaton (the length of a state is the maximum length of the target states of its outgoing transitions plus one).
- Given two states, the number of their outgoing transitions have to be also equal if the states are to be equivalent. This number is needed in order to construct the automaton correctly, and is also a relatively low number (ranging from 1 to the size of the alphabet used).
- Given two states, their weights have to be also equal if the states are to be equivalent. The weight is needed for the dictionary scheme, so it is a good idea to calculate it during the construction of the automaton (this is also possible since the weight of a state is the sum of the weights of the target states of its outgoing transitions).
  Of course, the range of possible values for the weight of a given state may be very high (ranging from 1 to the size of the lexicon), but empirical checks tell us that the most frequent weights are also relatively low numbers.

Therefore, our implementation of the register is a three-dimensional array which can be accessed by height, the number of outgoing transitions and weight. Each cell of this array contains the list of states that share this three features[6]. When a state is subject to being replaced or registered, we consider its features and it is only compared with the states in the corresponding list. Only then we verify the symbols of the labels of the outgoing transitions and their target states, which have to be equal if the states are to be equivalent.

When using our implementation of the incremental algorithm, the time needed to build the automaton of the Galena lexicon is reduced to 2.5 seconds. It takes an extra 4.6 seconds time to incorporate the information regarding tags, lemmas and probabilities, thus giving us a total compilation time of 7.1 seconds. In the case of the Erial lexicon, the equivalent times are $9.2 + 15.6 = 24.8$ seconds.

Finally, it should be noted that the recognition speed of our automata is around 80,000 words per second. This figure is also an improvement on that obtained when using [2], which reaches 35,000 words per second.

The only explanation we can find for this improvement is that we have also managed to produce a more efficient internal architecture for automata. The description of this internal representation lies outside the scope of this paper, but any requests for further information on this subject are welcome.

7 Conclusion

Through an in-depth study of the different methods for constructing acyclic finite-state automata, we have presented two main contributions for handling suitably large sets of words in the NLP domain. The first has been to design a general architecture for dictionaries, which is able to store the great amount

[6] This is actually only true for states with weights between 1 and 15, this being empirically the most frequents. States with greater weights are stored in a separate set of lists. Nevertheless, the lists in this latter set are also ordered by weight.

of lexical data related to the words. We have shown that it is the most compact representation when we need to deal with very specific information of these words such as probabilities, this scheme being particularly appropriate for stochastic NLP applications.

In a natural way, the second contribution completes our model of dictionaries by improving the incremental methods for constructing minimal acyclic automata. In incremental constructions, since parts of the dictionary that are already constructed (i.e. the states in the *register*) are no longer subject to future change, we can use other specific features of states in parallel. These features are sometimes inspired in the working mechanisms of our architecture for dictionaries (e.g. indexing *weights*) and sometimes in the base of other algorithms (e.g. *heights*). All of them allow us to improve the access to the registered parts and check equivalences with the new states very rapidly. In consequence, the total construction time of these minimal automata is less than that of those previous algorithms.

References

[1] Aho, A. V.; Sethi, R.; Ullman, J. D. (1985). Compilers: principles, techniques and tools. *Addison-Wesley*, Reading, MA. 135

[2] Daciuk, J.; Mihov, S.; Watson, B. W.; Watson, R. E. (2000). Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, vol. 26(1), pp. 3-16. 136, 141, 142, 143, 144

[3] Graña Gil, J. (2000). Robust parsing techniques for natural language tagging (in Spanish). *PhD. Thesis*, Departamento de Computación, Universidad de La Coruña (Spain). 140

[4] Hopcroft, J. E.; Ullman, J. D. (1979). Introduction to automata theory, languages and computations. *Addison-Wesley*, Reading, MA. 135, 140, 141

[5] Lucchesi, C. L.; Kowaltowski, T. (1993). Applications of finite automata representing large vocabularies. *Software - Practice and Experience*, vol. 23(1), pp. 15-30. 136, 138

[6] Revuz, D. (1992). Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, vol. 92(1), pp. 181-189. 136, 140, 141

[7] Watson, B. W. (1993). A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, (The Netherlands). 135

A Pseudo-Code of the Main Algorithms

We give in this appendix the figures with the details of all the algorithms cited in the paper.

```
function Index_to_Word (Index) =
    begin
        Current_State ← Initial_State;
        Number ← Index;
        Word ← Empty_Word;
        i ← 1;

        repeat
            for c ← First_Letter to Last_Letter do
                if (Valid_Transition (Current_State, c)) then
                    begin
                        Auxiliar_State ← Current_State[c];
                        if (Number > Auxiliar_State.Number) then
                            Number ← Number − Auxiliar_State.Number
                        else
                            begin
                                Word[i] ← c;
                                i ← i + 1;
                                Current_State ← Auxiliar_State;
                                if (Is_Final_State (Current_State)) then
                                    Number ← Number − 1;
                                exit forloop
                            end
                    end
        until (Number = 0);

        return Word
    end;
```

Fig. 4. Pseudo-code of function `Index_to_Word`

```
function Word_to_Index (Word) =
    begin
        Index ← 1;
        Current_State ← Initial_State;

        for i ← 1 to Length (Word) do
            if (Valid_Transition (Current_State, Word[i])) then
                begin
                    for c ← First_Letter to Predecessor (Word[i]) do
                        if (Valid_Transition (Current_State, c)) then
                            Index ← Index + Current_State[c].Number;
                    Current_State ← Current_State[Word[i]];
                end
            else
                return unknown word;

        if (Is_Final_State (Current_State)) then
            return Index
        else
            return unknown word
    end;
```

Fig. 5. Pseudo-code of function `Word_to_Index`

```
procedure Minimize_Automaton (Automaton) =
    begin
        Calculate Π;

        for i ← 0 to h(q0) do
            begin
                Sort the states of Πi by their transitions;
                Collapse all equivalent states
            end
    end;
```

Fig. 6. Pseudo-code of procedure `Minimize_Automaton`

```
function Incremental_Construction (Lexicon) =
 begin
    Register ← ∅;

    while (there is another word in Lexicon) do
        begin
          Word ← next word of Lexicon in lexicographic order;
          Common_Prefix ← Common_Prefix (Word);
          Last_State ← q0.Common_Prefix;
          Current_Suffix ← Word[(Length (Common_Prefix) + 1) ... Length (Word)];
          if (Has_Children (Last_State)) then
          Register ← Replace_or_Register (Last_State, Register);
          Add_Suffix (Last_State, Current_Suffix);
        end;

    Register ← Replace_or_Register (q0, Register);
    return Register
 end;
```

Fig. 7. Pseudo-code of function `Incremental_Construction`

```
function Replace_or_Register (State, Register) =
    begin
        Child ← Last_Child (State);
        if (Has_Children (Child)) then
            Register ← Replace_or_Register (Child, Register);
        if [(∃ q ∈ Q : q ∈ Register ∧ q ≡ Child)] then
            begin
                Last_Child (State) ← q;
                Delete (Child)
            end
        else
            [Register ← Register ∪ {Child};]
        return Register
    end;
```

Fig. 8. Pseudo-code of function `Replace_or_Register`

Bit Parallelism – NFA Simulation*

Jan Holub

Department of Computer Science and Engineering, Czech Technical University
Karlovo nám. 13, CZ-121 35, Prague 2, Czech Republic
holub@fel.cvut.cz

Abstract. This paper deals with one of possibilities of use of nondeterministic finite automaton (*NFA*)—simulation of *NFA* using the method called bit parallelism. After a short presentation of the basic simulation method, the bit parallelism is presented on one of the pattern matching problems. Then a flexibility of the bit parallelism is demonstrated by a simulation of *NFA*s for other pattern matching problems.

1 Introduction

In Computer Science there is a class of problems that can be solved by finite automata. For some of these problems one can construct directly a deterministic finite automaton (*DFA*) that solves them. For other problems it is easier to build a nondeterministic finite automaton (*NFA*). Since one cannot use *NFA* directly because of its nondeterminism, one should transform it to the equivalent *DFA* using the standard subset construction [HU79, Koz97] or one should simulate a run of the *NFA* using one of the simulation methods [Hol00].

When transforming *NFA*, one can get the *DFA* with a huge amount of states (up to $2^{|Q_{NFA}|}$, where $|Q_{NFA}|$ is the number of states of *NFA*). The time complexity of the transformation is proportional to the number of states of *DFA*. The run is then very fast (linear with the length of an input text). On the other hand, when simulating the run of *NFA*, the time and space complexities are given by the number of states of *NFA*. The run of the simulation is then slower.

There are known three simulation methods [Hol00]: *basic simulation method*, *dynamic programming*, and *bit parallelism*. All of these methods use breadth-first search for traversing the state space. The first overview of the simulation methods was presented in [HM99].

At the beginning we will shortly introduce the basic simulation method, which is the base for other simulation methods, and its bitwise implementation. Then we will present the method called bit parallelism. We show, how the bit parallelism can be adjusted to various *NFA*s of the exact and approximate pattern matching. This simulation is very efficient for *NFA*s with a regular structure, when a lot of transitions can be executed at once using bitwise operations.

* Partially supported by the GAČR grants 201/98/1155, 201/01/1433, and 201/01/P082.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 149–160, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Definitions

Let Σ be a nonempty input alphabet, Σ^* be the set of all strings over Σ, ε be the *empty string*, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. If $w \in \Sigma^*$, then $|w|$ denotes the *length* of w ($|\varepsilon| = 0$). If $a \in \Sigma$, then $\overline{a} = \Sigma \setminus \{a\}$ denotes a *complement* of a over Σ. If $w = xyz$, $x, y, z \in \Sigma^*$, then x, y, z are *factors* (substrings) of w, moreover, x is a *prefix* of w and z is a *suffix* of w.

Deterministic finite automaton (*DFA*) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is a set of input symbols, δ is a mapping (transition function) $Q \times \Sigma \mapsto Q$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. We extend δ to a function $\hat{\delta}$ mapping $Q \times \Sigma^+ \mapsto Q$. *Terminal state* denotes a state $q \in Q$ that has no outgoing transition (i.e., $\forall a \in \Sigma$, $\delta(q, a) = \emptyset$ or using $\hat{\delta}$: $\forall u \in \Sigma^+, \hat{\delta}(q, u) = \emptyset$).

Nondeterministic finite automaton (*NFA*) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q, Σ, q_0, F are the same like in *DFA* and δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto 2^{|Q|}$. We also extend δ to $\hat{\delta}$ mapping $Q \times \Sigma^* \mapsto 2^{|Q|}$. *DFA* (resp. *NFA*) accepts a string $w \in \Sigma^*$ if and only if $\hat{\delta}(q_0, w) \in F$ (resp. $\hat{\delta}(q_0, w) \cap F \neq \emptyset$).

If $P \subseteq Q$, then for *NFA* we define $\varepsilon CLOSURE(P) = \{q' \mid q' \in \hat{\delta}(q, \varepsilon), q \in P\} \cup \{P\}$. An *active state* of *NFA*, when the last symbol of a prefix w of an input string is processed, denotes each state q, $q \in \hat{\delta}(q_0, w)$. At the beginning, only q_0 is an active state.

An algorithm $\mathcal{A}$ *simulates a run of an NFA*, if $\forall w$, $w \in \Sigma^*$, it holds that $\mathcal{A}$ with given w at the input reports all information associated with each final state q_f, $q_f \in F$, after processing w, if and only if $q_f \in \hat{\delta}(q_0, w)$.

3 Basic Simulation Method

The basic simulation method maintains a set S of active states during the whole simulation process. At the beginning only the state q_0 is active and then we evaluate ε-transitions leading from q_0: $S_0 = \varepsilon CLOSURE(\{q_0\})$. In the i-th step of the simulation with text $T = t_1 t_2 \dots t_n$ on input (i.e., t_i is processed), we compute a new set S_i of active states from the previous set S_{i-1} as follows: $S_i = \bigcup_{q \in S_{i-1}} \varepsilon CLOSURE(\delta(q, t_i))$. In each step we also check, whether $S_i = \emptyset$, then the simulation finishes (i.e., *NFA* does not accept T), and whether $S_i \cap F \neq \emptyset$, then we report, that a final state is reached (i.e., *NFA* accepts string $t_1 t_2 \dots t_i$). If each final state has an associated information, we report it as well.

Note, that each configuration of set S determines one state of the equivalent *DFA*. If we store each such configuration, we could get a transformation of *NFA* to *DFA*, but with the advantage that we would compute only used transitions and states. It is possible to combine the simulation and the transformation. In such case, we would have some 'state-cache', where we store some limited number of used configurations and label them by deterministic states. We would store also used transitions of the stored states. If we should execute then a transition that is already stored together with its destination state, we would use just the

corresponding deterministic label instead of computing the whole set S. This is obviously faster than computing the corresponding configuration.

We implement this simulation by using bit-vectors as described in [Hol00]. This implementation runs in time $\mathcal{O}(n|Q|\lceil\frac{|Q|}{w}\rceil)$ and space $\mathcal{O}(|\Sigma||Q|\lceil\frac{|Q|}{w}\rceil)$, where w is a length of used computer word in bits, $|Q|$ is a number of states of *NFA*, and n is a length of the input string.

4 Bit Parallelism

The bit parallelism is a method that uses bit vectors and benefits from the feature that the same bitwise operations (OR, AND, ADD, ... etc.) over groups of bits (or over individual bits) can be performed at once in parallel way over the whole bit vector. The representatives of the bit parallelism are Shift-Or, Shift-And, and Shift-Add algorithms. We use only Shift-Or algorithm in this paper.

The algorithms, that use the bit parallelism, were developed without the knowledge that they simulate *NFA* solving the given problem. At first an algorithm using the bit parallelism was used for the exact string matching (Shift-And in [Döm64]), the multiple exact string matching (Shift-And in [Shy76]), the approximate string matching using the Hamming distance (Shift-Add in [BYG92]), the approximate string matching using the Levenshtein distance (Shift-Or in [BYG92] and Shift-And in [WM92]) and for the generalized pattern matching (Shift-Or in [Abr87]), where the pattern consists not only from symbols but also from sets of symbols.

The simulation using the bit parallelism [Hol96b] will be shown on the *NFA* for the approximate string matching using the Levenshtein distance. This problem is defined as a searching for all occurrences of pattern $P = p_1p_2 \ldots p_m$ in text $T = t_1t_2 \ldots t_n$, where the found occurrence X (substring of T) can have

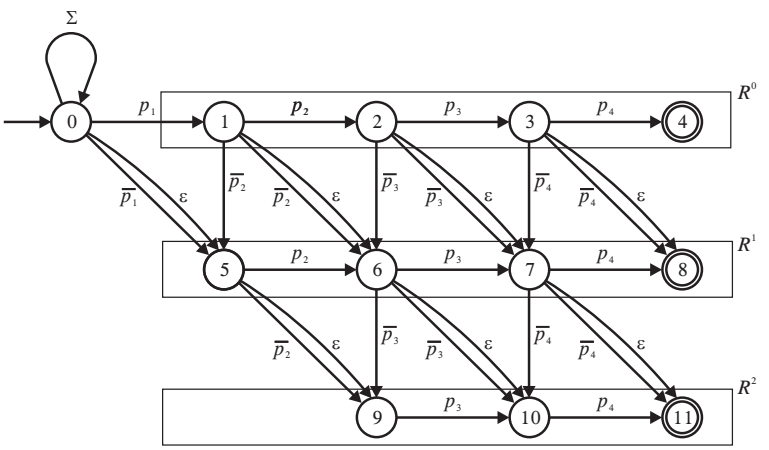


Fig. 1. Bit parallelism uses one bit vector R for each level of states of *NFA*

at most k differences. The number of differences is given by the Levenshtein distance $D_L(P, X)$, which is defined as the minimum number of edit operations *replace*, *insert*, and *delete*, that are needed to convert P to X.

Figure 1 shows the *NFA* constructed for this problem ($m = 4$, $k = 2$). The horizontal transitions represent matching, the vertical transitions represent *insert*, the diagonal ε-transitions represent *delete*, and the remaining diagonal transitions represent *replace*. The self-loop of the initial state provides skipping the prefixes of T located in front of the occurrences.

Shift-Or algorithm uses for each level (row) l, $0 \leq l \leq k$, of states one bit vector R^l (of size m). Each state of the level is then represented by one bit in the vector. If a state is active, then the corresponding bit is 0, if it is not active, the bit is 1. We have no bit representing q_0, since this state is always active. Formula 1 shows, how the vectors $R_i^l = [r_{1,i}^l, r_{1,i}^l \dots r_{m,i}^l]$ in the i-th step are computed.

$$
\begin{array}{lll}
r_{j,0}^l \leftarrow 0, & & 0 < j \leq l, 0 \leq l \leq k \\
r_{j,0}^l \leftarrow 1, & & l < j \leq m, 0 \leq l \leq k \\
R_i^0 \leftarrow \mathbf{shr}(R_{i-1}^0) \text{ OR } D[t_i], & & 0 < i \leq n \\
R_i^l \leftarrow (\mathbf{shr}(R_{i-1}^l) \text{ OR } D[t_i]) & /* \textit{ match } */ & \\
\quad \text{AND } (\mathbf{shr}(R_{i-1}^{l-1}) \text{ OR NOT}(D[t_i])) & /* \textit{ replace } */ & \\
\quad \text{AND } \mathbf{shr}(R_i^{l-1}) & /* \textit{ delete } */ & \\
\quad \text{AND } (R_{i-1}^{l-1} \text{ OR NOT}(\mathbf{shl}(D[t_i])) \text{ OR } V), & /* \textit{ insert } */ & \\
 & & 0 < i \leq n, 0 < l \leq k
\end{array}
\tag{1}
$$

In the formula, **shr**() (resp. **shl**()) is the bitwise operation right (resp. left) shift, OR (resp. AND) is the bitwise operation OR (resp. AND), and NOT() is a bitwise operation that exchanges 0s and 1s.

At the beginning only the states of $\varepsilon CLOSURE(\{q_0\})$ are active, therefore the first l bits are 0 in each vector R_0^l. The transitions representing matching are the only transitions leading to the states of level 0. To simulate these transitions, we shift vector R_{i-1}^l to the right[1]—it is represented by term $\mathbf{shr}(R_{i-1}^l)$. The operation **shr**() inserts 0 at the beginning of the vector, that implements the self-loop of the initial state. At this moment, all active states moved to the right and we have to select only those transitions, that are labeled by t_i. For this selection we use table D of mask vectors. The table D is defined as follows: $D[x] = [d_{1,x} d_{2,x} \dots d_{m,x}]$, $x \in \Sigma$, where $d_{j,x} = 0$, if $p_j = x$, or 1, otherwise. When we execute the bitwise operation OR over the shifted vector and mask vector $D[t_i]$, 1s are inserted to the positions, where the transitions are not labeled by t_i. Thus 0 remains only on such a position that the previous position contained 0 and there is a transition match labeled by t_i connecting these two positions.

Term $\mathbf{shr}(R_{i-1}^{l-1})$ OR NOT$(D[t_i])$ represents *replace*. In this case we shift the vector from the previous level $l - 1$. Then we have to select the transitions

[1] In practice, Shift-Or algorithm has the right and left shifts exchanged because of easier implementation when the vectors are too long and have to be divided into several computer words.

labeled by t_i. While in transition match, these transitions correspond to the positions matching the pattern, in case of transition *replace*, these transitions correspond to the positions mismatching the pattern. Therefore we use mask vector NOT$(D[t_i])$.

Term **shr**(R_i^{l-1}) represents *delete*. In this case we shift the new value of the vector from the previous level $l-1$. We do not use any mask vector, since we implement ε-transitions, that apply always. We use the new value of R^{l-1}, since no input symbol is read when ε-transitions are executed.

Term R_{i-1}^{l-1} OR NOT(**shl**$(D[t_i])$) OR V represents *insert*. We take the previous value of vector R^{l-1}. Since each such transition is labeled by $\overline{p}_j$ (i.e., mismatching the label of the transition match leading from the same state), we use mask vector $D[t_i]$. We have to exchange 0s and 1s in the vector, but in addition we have to shift this vector in the opposite direction, than in which we usually shift the vectors. It is because we do not shift vector R_{i-1}^{l-1} (it is a vertical transition that does not change a position in the pattern), we must get the corresponding bits to the correct positions in this way. Vector $V = [00\ldots01]$ is used to prevent *insert* transitions leading into the final states.

All these terms are connected by bitwise operation AND, which adds up all states arriving along all the transitions leading to the same state. If the last bit of vector R^l is 0, the final state of level l is active. In this case it means that the pattern has been found with at most l differences.

Example:
Let $P = adbbca$, $T = adcabcaabadbbca$, and $k = 3$. Table D is shown in Table 1 and Table 2 shows the simulation process of *NFA* run.

5 Adjusting Bit Parallelism

Now, we show how the bit parallelism can simulate other *NFA*s. In the approximate string matching using the Hamming distance, the only allowed edit operation is *replace*. By omitting transitions *insert* and *delete* from the *NFA* for the approximate string matching using the Levenshtein distance we get the *NFA* for the approximate string matching using the Hamming distance. By omitting terms simulating *insert* and *delete* transitions in Formula 1 we get the formula

Table 1. Matrix D for pattern $P = adbbca$

D	a	b	c	d	$\Sigma \setminus \{a,b,c,d\}$
a	0	1	1	1	1
d	1	1	1	0	1
b	1	0	1	1	1
b	1	0	1	1	1
c	1	1	0	1	1
a	0	1	1	1	1

Table 2. Matrices R^l for the approximate string matching using the Levenshtein distance ($\overline{p}$ version, $P = adbbca$, $k = 3$, and $T = adcabcaabadbbca$)

R^0	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	1	**0**	1	1	**0**	1	1	**0**	**0**	1	**0**	1	1	1	1	**0**
d	1	1	**0**	1	1	1	1	1	1	1	1	**0**	1	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**

R^1	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	1	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	1	1	1	**0**
b	1	1	**0**	**0**	1	**0**	1	1	1	**0**	1	**0**	**0**	1	1	1
b	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	1	1
c	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**	1
a	1	1	1	1	1	1	1	1	1	1	1	1	1	1	**0**	**0**

R^2	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**
b	1	1	**0**	**0**	**0**	**0**	**0**	1	1	**0**	**0**	**0**	**0**	1	1
c	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**	1
a	1	1	1	1	**0**	1	1	**0**	1	1	1	1	1	**0**	**0**	**0**

R^3	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
d	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
c	1	1	**0**	**0**	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	1
a	1	1	1	**0**	**0**	1	**0**	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**

for simulating the *NFA* for the approximate string matching using the Hamming distance.

If we use only vector R^0 (i.e., $k = 0$), we get the *NFA* for the exact string matching. In this case the equivalent *DFA* has the same number of states [Hol00], therefore the simulation is less efficient than the transformation to the equivalent *DFA* and its run. There is also a direct construction of this *DFA*.

If we label *replace* and *insert* transitions by Σ (i.e., the transitions are used even if t_i is the matching symbol), we get so called Σ version of *NFA* [Hol00]. This *NFA* has a little simpler formula, since we do not need any mask vector for the simulation of *replace* and *insert* transitions. On the other hand, when an exact occurrence is found, then this *NFA* reports, in addition, the approximate occurrences with $1, 2, \ldots, k$ differences following the exact occurrence. The original

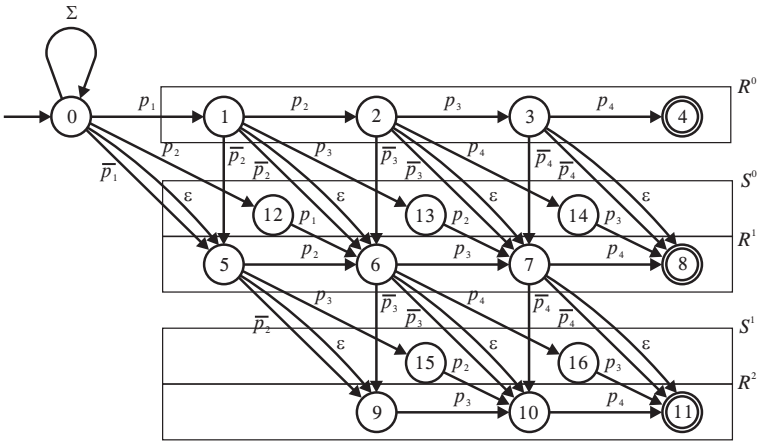


Fig. 2. *NFA* for the approximate string matching using the generalized Levenshtein distance

Shift-Or algorithm for the approximate string matching using the Levenshtein distance [WM92] simulates the Σ version of *NFA*.

If we introduce transition *transpose* (two adjacent symbols are exchanged) into the approximate string matching using the Levenshtein distance, we get the approximate string matching using the generalized Levenshtein distance [Hol97]. Since the transition *transpose* reads two symbols, it requires one auxiliary state. The *NFA* for the approximate string matching using the generalized Levenshtein distance is shown in Figure 2.

For the auxiliary states we have to introduce k new bit-vectors each of size m. The formula for the simulation of this *NFA* is shown in Formula 2.

$$
\begin{array}{lll}
r^l_{j,0} \leftarrow 0, & & 0 < j \leq l, 0 < l \leq k \\
r^l_{j,0} \leftarrow 1, & & l < j \leq m, 0 \leq l \leq k \\
R^0_i \leftarrow \mathbf{shr}(R^0_{i-1}) \text{ OR } D[t_i], & & 0 < i \leq n \\
R^l_i \leftarrow (\mathbf{shr}(R^l_{i-1}) \text{ OR } D[t_i]) & & /* \textit{ match } */ \\
\quad \text{AND } (\mathbf{shr}(R^{l-1}_{i-1}) \text{ OR NOT}(D[t_i])) & & /* \textit{ replace } */ \\
\quad \text{AND } \mathbf{shr}(R^{l-1}_i \text{ AND } (S^{l-1}_{i-1} \text{ OR } D[t_i])) & & /* \textit{ delete} \text{ and } \textit{transpose } */ \\
\quad \text{AND } (R^{l-1}_{i-1} \text{ OR NOT}(\mathbf{shl}(D[t_i])) \text{ OR } V), & & /* \textit{ insert } */ \\
 & & 0 < i \leq n, 0 < l \leq k \\
s^l_{j,0} \leftarrow 1, & & 0 < j \leq m, 0 \leq l < k \\
S^l_i \leftarrow \mathbf{shr}(R^l_{i-1}) \text{ OR } \mathbf{shl}(D[t_i]), & & 0 < i < n, 0 \leq l < k
\end{array}
\tag{2}
$$

Now, we show the simulation of transitions *transpose* leading to state q of l-th level. Let the transition match leading to q is labeled by p_j. The *transpose* transition leads from state q' of level $l-1$. The term $\mathbf{shr}(R^l_{i-1})$ OR $\mathbf{shl}(D[t_i])$ represents the transition leading from state q' to an auxiliary state q''. This transition is labeled by symbol p_j, while the match transition leading from q'

is labeled by p_{j-1}, therefore we have to shift the mask vector $D[t_i]$ to the left. The term $\mathbf{shr}(S_{i-1}^{l-1}$ OR $D[t_i])$ then represents the transition leading from q'' to q. Since this transition is labeled by p_{j-1}, we have to shift to the right also the mask vector.

We get the exact and approximate sequence matching from the exact and approximate string matching, if we allow any number of symbols to be located between two adjacent symbols of the pattern. The *NFA* for the sequence matching we get from the *NFA* for the string matching just by inserting self-loop to each nonfinal state. Such self-loop of state q is then labeled by $\overline{p}_j$, where p_j is a label of the match transition leading from q [Hol97]. To get the formula for sequence matching, we should insert just term AND $(R_{i-1}^l$ OR $\mathbf{shl}($NOT$(D[t_i]))$ OR $V)$, which simulates the self-loop, into the formula for the corresponding string matching task. The term represents that we keep the previous value of the vector and select the states that have outgoing match transition labeled by a symbol mismatching t_i. We shift the mask vector to the left, otherwise we would select the states that have incoming (not outgoing) match transition labeled by a symbol mismatching t_i.

If we do not care about the exact number of errors in the found string and we only want to be sure that the number of errors is less or equal to k, we can use the reduced *NFA* [Hol96a]. The reduced *NFA* for the approximate string matching contains in each level only the first $(m - k + 1)$ states. Other states are removed, since they are needed only to determine the exact number of errors in the found string. For the simulation of this *NFA*, we can use shortened bit vectors (of size $(m - k + 1)$ for the Hamming distance and of size $(m - k)$ for the Levenshtein and generalized Levenshtein distances). The formulae for the simulation are similar like for the nonreduced *NFA*s. We only have to change some shifts, since j-th bit in each vector R^l represents the state corresponding to the $(j + l)$-th position in the pattern (and not j-th position like in case of nonreduced *NFA*s). For details see [Hol96b, Hol96a, Hol00].

The string and sequence verification is a task, where we verify whether the pattern is located at the beginning of an input string—no string can be located in front of the occurrence of the pattern. We get the *NFA* for the verification, if we remove the self-loop of the initial state in the *NFA* for string matching. Since the self-loop of the initial state is simulated by the insertion of 0s at the beginning of bit vector within shift operation, we get the formula simulating the *NFA* for the verification, if we prevent these insertions in the formula simulating the *NFA* for the matching [Hol00].

Weighted approximate string and sequence matching introduces for each edit operation (*replace*, *insert*, *delete*, and *transpose*) its weight (w_R, w_I, w_D, and w_T respectively). It can be either independent on the symbol being edited or it can vary according to the symbol. In the *NFA* it is represented in such a way that transitions representing various edit operations lead to states in various levels according to their weights. In formulae for simulating *NFA* it is represented in such a way that when computing R_i^l, terms does not contain R_{i-1}^{l-1} and S_{i-1}^{l-1} but $R_{i-1}^{l-w_R}$, $R_{i-1}^{l-w_I}$, $R_i^{l-w_D}$, and $S_{i-1}^{l-w_T}$. For details see [Hol00].

In [HIMM99] there is also shown the simulation of *NFA* for the distributed pattern matching where the pattern and/or the input text can be either "singular" (ordinary) strings or "multiple" ones (strings distributed in several lines).

6 Other Methods of Bit Parallelism

In the previous sections we described only Shift-Or algorithm. If we exchange the meaning of 0s and 1s and the usage of ANDs and ORs in the formulae presented, we get Shift-And algorithm [WM92].

Other method of bit parallelism is Shift-Add algorithm [BYG92]. In this algorithm we have one bit-vector, which contains m blocks of bits of size $b = \lceil \log_2 m \rceil$ (one block for each depth of *NFA* for the approximate string matching using the Hamming distance). The formula for computing such vector then consists of shifting the vector by b bits and adding with the mask vector for current input symbol t. This vector contains 1 in $(b.j)$-th position, if $t \neq p_{j+1}$, or 0, elsewhere. This algorithm runs in time $\mathcal{O}(\lceil \frac{mb}{w} \rceil n)$.

We can also use Shift-Add algorithm for the weighted approximate string matching using the Hamming distance. In such case each block of bits in mask vector contains binary representation of weight of the corresponding edit operation *replace*. We have also to enlarge the length of block of bits to prevent a carry to the next block of bits. Note, that Shift-Add algorithm can also be considered as an implementation of other simulation method called dynamic programming.

In [BYN96b, BYN96a] they show an algorithm that simulates reduced *NFA* for the approximate string matching using the Levenshtein distance by ε-diagonals in time $\mathcal{O}(\lceil \frac{km}{w} \rceil n)$. They use one bit-vector with $(m - k)$ blocks of size $(k + 1)$. Each ε-diagonal is represented by one block.

In [BYN96a] they improve the approximate string matching using the Levenshtein distance in such a way, that they search for any of the first $(k + 1)$ symbols of the pattern. If they find any of them, they start *NFA* simulation and if the simulation then reaches the initial situation (i.e., only the states located in ε-diagonal leading from the initial state are active), they again start searching for any of the first $(k + 1)$ symbols of the pattern, which is faster than the simulation.

Shift-Or algorithm can also be used for the exact string matching with don't care symbols, classes of symbols, and complements as shown in [BYG92]. In [WM92] they extend this method to unlimited wild cards and apply the above features for the approximate string matching. They also use bit parallelism for regular expressions, for the weighted approximate string matching, for set of patterns, and for the situations, when errors are not allowed in some parts of pattern (another mask vector is used).

All the previous cases have good time complexity if the *NFA* simulation fits in one computer word (each vector in one computer word). If *NFA* is larger, it is necessary to partition *NFA* or pattern as described in [BYN96b, BYN96a, BYN97], [BYN99, NBY98, WM92].

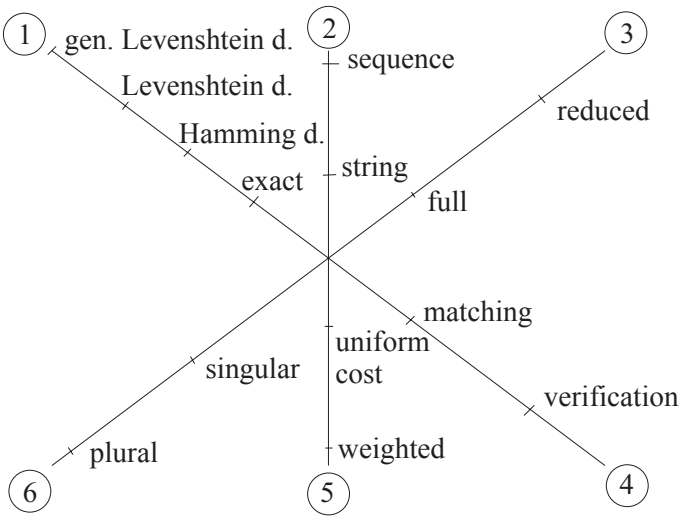


Fig. 3. 6D classification of the problems described in this paper

Shift-Or algorithm can also be used for multiple pattern matching [BYG92], [BYN97] or for the distributed pattern matching [HIMM99].

7 Conclusion

In this paper we have presented the basic simulation method and the bit parallelism. The latter was presented on *NFA*s for many problems that can be classified by 6-dimensional space in Figure 3[2].

According to the first dimension we are interested in an occurrence of pattern without any error or with a specified number of errors given by Hamming, Levenshtein, and generalized Levenshtein distances. The second dimension consider whether the pattern is a string or a sequence. We can decide, according to the third dimension, whether we are interested in an exact number of errors in the found occurrence (while the number is under some limit) or not (then we can use reduced *NFA*). The fourth dimension expresses, whether we perform verification or matching. The fifth dimension takes into account that some of edit operations can have different weights. The sixth dimension considers the distributed pattern matching, where the pattern and/or text can be distributed in several lines.

Simulation methods are used for *NFA*s, when the corresponding *DFA*s are too complex to be fit in the main memory or when the preprocessing time for *DFA* is too high comparing with the time of run (i.e., it is not efficient).

[2] Note, that dimensions 3 and 5 make no sense, when we do not allow any error in the pattern.

The time and space analysis of the Shift-Or algorithm is as follows [WM92]. Denote the computer word size by w. The preprocessing requires $\mathcal{O}(m|\Sigma|)$ time plus $\mathcal{O}(k\lceil\frac{m}{w}\rceil)$ to initialize the k vectors. The running time is $\mathcal{O}(nk\lceil\frac{m}{w}\rceil)$. The space complexity is $\mathcal{O}(|\Sigma|\lceil\frac{m}{w}\rceil)$ for mask matrix D plus $\mathcal{O}(k\lceil\frac{m}{w}\rceil)$ for the k vectors.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987. 151

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992. 151, 157, 158

[BYN96a] R. A. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 3rd South American Workshop on String Processing*, pages 47–63, Recife, Brazil, 1996. Carleton University Press. 157

[BYN96b] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, number 1075 in Lecture Notes in Computer Science, pages 1–23, Laguna Beach, CA, 1996. Springer-Verlag, Berlin. 157

[BYN97] R. A. Baeza-Yates and G. Navarro. Multiple approximate string matching. In F. K. H. A. Dehne, A. Rau-Chaplin, J.-R. Sack, and R. Tamassia, editors, *Proceedings of the 5th Workshop on Algorithms and Data Structures*, number 1272 in Lecture Notes in Computer Science, pages 174–184, Halifax, Nova Scotia, Canada, 1997. Springer-Verlag, Berlin. 157, 158

[BYN99] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. 157

[Döm64] B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, (3):29–46, 1964. 151

[HIMM99] J. Holub, C. S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed string matching using finite automata. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 114–128, Perth, WA, Australia, 1999. 157, 158

[HM99] J. Holub and B. Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In J.-M. Champarnaud, D. Maurel, and D. Ziadi, editors, *Proceedings of the 3rd International Workshop on Implementing Automata'98*, number 1660 in Lecture Notes in Computer Science, pages 92–99, Rouen, France, 1999. Springer-Verlag, Berlin. 149

[Hol96a] J. Holub. Reduced nondeterministic finite automata for approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 19–27, Czech Technical University, Prague, Czech Republic, 1996. Collaborative Report DC–96–10. 156

[Hol96b] J. Holub. Approximate string matching in text. Master's thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic, February 1996. 151, 156

[Hol97] J. Holub. Simulation of NFA in approximate string and sequence matching. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 39–46, Czech Technical University, Prague, Czech Republic, 1997. Collaborative Report DC–97–03. 155, 156

[Hol00] J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching*. Ph. D. Thesis, Czech Technical University, Prague, Czech Republic, 2000. 149, 151, 154, 156

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations*. Addison-Wesley, Reading, MA, 1979. 149

[Koz97] D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, 1997. 149

[NBY98] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, University of Chile, 1998. 157

[Shy76] R. K. Shyamasundar. A simple string matching algorithm. Technical report, Tata Institute of Fundamental Research, 1976. 151

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992. 151, 155, 157, 159

Improving Raster Image Run-Length Encoding Using Data Order

Markus Holzer[1] and Martin Kutrib[2]

[1] Institut für Informatik, Technische Universität München
Arcisstraße 21, D-80290 München, Germany
holzer@informatik.tu-muenchen.de
[2] Institut für Informatik, Universität Gießen
Arndtstraße 2, D-35392 Gießen, Germany
kutrib@informatik.uni-giessen.de

Abstract. We examine the technique of run-length encoding in combination with data order, where our attention is focused on good performance of image operations such as, e.g., rotation, reflection, and zooming. To this end we develop a new type of data order that supports these operations well and allows to perform them on a variant of a double-queue automaton directly on the compressed data stream. Because of its shape we call this data order shamrock or S-order. To confirm our theoretical results on S-order we have performed some experiments on sample data using various data orderings that appear in the literature.

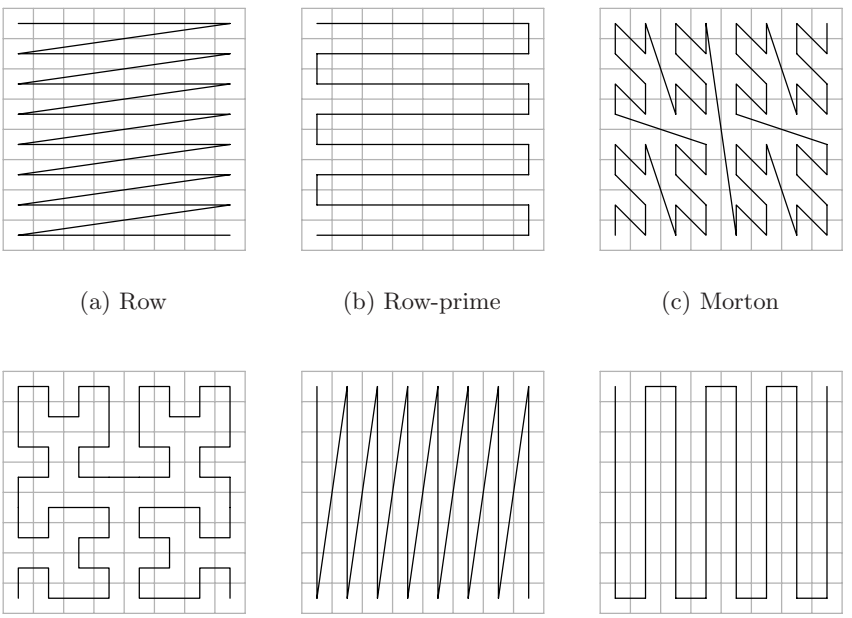
1 Introduction

Run-length encoding (RLE) is one of the most simplest lossless type of data compression schemes and is based on a simple principle of encoding data; see, e.g., Salomon [9]. This principle is that every partial stream which is formed of the same data values, i.e., sequence of repeated data values, is replaced by a single value and a count number—in the terminology of data compression repeating values are called a *run*. This intuitive principle works best on certain data types in which sequences of repeated data values can be noticed. Therefore RLE is usually applied to files that contain large numbers of consecutive occurrences of the same byte pattern. Good examples are text files which contain multiple spaces for indentation and formatting paragraphs, tables, and charts, as well as digitized signals as, e.g., monochrome images or sound files. Although RLE can be easily implemented and quickly executed it cannot achieve great compression ratios.

Many researchers, including many in the database community, have proposed clustering data using orderings to improve compression ratios. Since the computation of the stream, which can be compressed using the above described principle, is the heart of RLE, it is natural to pay special attention to these data orderings. One of the most common form of grided data representations as, e.g., raster images, is row order, where cells are traversed in sequence by row and, within row, by column. That RLE may benefit from using different

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 161–176, 2002.
© Springer-Verlag Berlin Heidelberg 2002

data ordering was shown by several authors, especially geographers. One of the first experiments with data orderings are often credited to Morton [8] in the mid 1960's. Later further research into this topic was undertaken by Goodchild and Grandfield [5]. In this paper four data orders were tested to determine their compression capability in combination with RLE. Besides the already mentioned row ordering also row-prime, Morton order, and Hilbert order—see Figures 1(a)–(d)—were considered. Note that Morton order is also called Z-order and plays a crucial role in spatial databases to cluster multi-dimensional data; for further details we refer to Gaede and Günther [4]. It turned out, that storage could be reduced up to 60% using Hilbert order, and 25% using Z-order, as opposed to using row order, for (monochrome) images with high spatial homogeneity. For images with low spatial homogeneity the tests resulted in a 5% reduction from Hilbert order and a 5% increase from Z-order, over row order. Similar results were obtained by a comparative analysis by Abel and Mark [1]. Intuitively, these results tell us that using a Hilbert scan the coherence will be improved by avoiding the discontinuities found at the end of row order scan lines; adjacent pixels from a Hilbert scan share the same area because of its fractal nature, and are thus more likely to be similar than pixels in horizontally scanned lines. These factors improve the performance of RLE.



(a) Row (b) Row-prime (c) Morton

(d) Hilbert (e) Column (f) Column-prime

Fig. 1. Some standard data orderings

Digital image compression deals with the problem of finding such a succinct representation for raster images. Over the years quite a few compression techniques have been developed that aim to eliminate redundancy from raster images; see, e.g., Kou [7]. Images are preferably maintained in compressed form not only to save storage space, but also to perform operations faster on the compressed image. This is a realistic objective because in compressed form less data has to be processed. Because it must be expected that different properties of the various orderings are reflected in differing degrees of suitability for various tasks we examine the technique of RLE in combination with data order, where our attention is focused on good performance of image operations such as, e.g., rotation, reflection, and zooming.

Intuitively speaking, a data order should possess certain desirable properties: (1) Pass once through each point in the image, (2) easy description and computation of the order, (3) be stable when the image becomes larger, and (4) provide variable resolution. For our purpose we have to develop a new sort of data order which supports these image operations, since non of the existing orders are useful. Because of its shape we call it shamrock or S-order, for short. More precisely we will see that the support of the image operations rotation and reflection clashes with condition (1). Hence, S-order must access certain image points more than once, which might be some disadvantage compared to Hilbert order. Nevertheless, our experiments show that S-order is still comparable with Z-order and mostly dominates it. Moreover, the S-order can be computed by a recursive algorithm as in the case of the Hilbert order and the order is actually so simple that the mentioned image operations can be performed by a variant of a double-queue-automaton.

The paper is organized as follows. In the next section we introduce the necessary notations. Section 3 introduces the new sort of data order under the light that non of the existing orderings support rotation and reflection well. Then in Section 4 we briefly describe some algorithms performing raster image operations on the encoded data in linear time with respect to the input size. Finally, Section 5 summarizes experiments, which we have performed recently.

2 Definitions

In this section we give the necessary notations on raster images and data order. Let n be a power of two. An $n \times n$ raster image is defined to be a quadratic array or grid of same size built from regularly sampled values, known as pixels—picture elements. In a raster image each pixel has a coordinate (i, j), with $1 \leq i, j \leq n$, and a number associated with it, generally specifying a color which the pixel should be displayed in. The pixels with coordinates $(1, 1)$ and (n, n), respectively, lie in the lower left and upper right corner, respectively. Throughout the paper we only make use of monochrome raster images, hence restricting the colors to *black* and *white*.

The image operations rotation and reflection are nothing other than certain permutations of the pixels in the raster image. Let d refer to the $\frac{\pi}{2}$ rotation

around the center C of the image, and s to the reflection at the vertical line intersecting the center—see Figures 2(a) and (b). Then d and s are permutations operating on the set of pixel coordinates $P_n = \{ (i,j) \mid 1 \leq i,j \leq n \}$, satisfying $d^4 = id$, $s^2 = id$, and $dsd = s$. From these equations one sees that d and s generate the dihedral group D_4 containing

$$D_4 = \{id, d, d^2, d^3, s, sd, sd^2, sd^3\}.$$

Observe, that D_4 has order 8. For further information on dihedral groups we refer to Weinstein [10]. The four axis of symmetry induced by D_4 are shown in Figure 2(c). We say that a raster image is closed under the operations of the dihedral group D_4 if and only if the picture is unchanged by the transformations in D_4.

Finally, we make our intuition on data orderings more precise. The definition of a data order reads as follows. Again, let n be a natural number. A *data order* φ on an $n \times n$ grid is a one-to-one mapping

$$\varphi : \{1, \ldots, n^2\} \to \{1, \ldots, n\} \times \{1, \ldots, n\}.$$

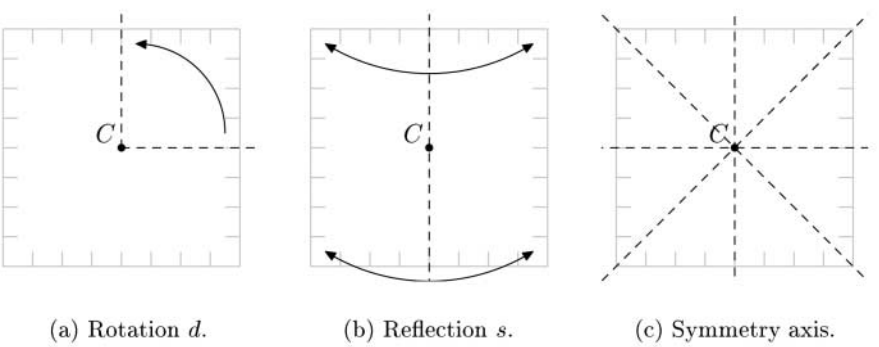
A data order φ is called *continuous* if and only if $d_2(\varphi(i), \varphi(i+1)) = 1$ holds for all $1 \leq i < n^2$, where d_2 denotes the Euclidean distance d_2, and a continuous data order φ is *cyclic* if and only if $d_2(\varphi(1), \varphi(n^2)) = 1$. We associate an undirected graph $D(\varphi)$ with a data order φ in the obvious way, i.e., $D(\varphi) = (V, E)$, where

$$V = \{1, \ldots, n\} \times \{1, \ldots, n\}$$

and

$$E = \{ (\varphi(i), \varphi(i+1)) \mid 1 \leq i < n^2 \} \cup \{(\varphi(n^2), \varphi(1))\}.$$

We consider two data orders two be equal if and only if they induce the same graph. This is particularly useful because, e.g., data order φ and the data order



(a) Rotation d. (b) Reflection s. (c) Symmetry axis.

Fig. 2. Basics on the dihedral group D_4

where every image point is moved one step along the underlying graph or in other words

$$\varphi'(i) = \begin{cases} \varphi(i+1) \text{ if } 1 \leq i < n^2 \\ \varphi(1) \qquad \text{otherwise,} \end{cases}$$

are equivalent. Thus, the starting point of a data order is not important. Moreover, if φ is continuous, then so is φ'. The some holds in case of cyclic data orders.

A data order φ is *closed* under the dihedral group D_4 if and only if the graph $D(\varphi)$ remains unchanged by the transformations in D_4, i.e., if one applies the appropriate permutation operation on the edges. Note, that none of the data orderings shown in Figure 1 are closed under the dihedral group D_4.

Observe, that a continuous data order traverses a grid making unit steps and turning only at right angles. Sometimes a continuous data order in our sense is called *discrete space-filling curve* in the literature. We have already seen the most typical example of a continuous data ordering, namely Hilbert ordering, which is based on the equally named space-filling curve [6]. It is created by starting with an initial shape, looking like a *staple*, that is copied and rotated four times with connecting lines inserted to fill a square area. The first few stages are shown in Figure 3.

Its simplicity and beauty derive from the fact that it progressively subdivides a square array down into an array of four sub-squares. The final curve is created by repeating and rotating the copying infinitely. This is yet another feature of continuous data orderings since they are usually *self-similar*, i.e., the data order or curve can be generated by putting together identical units, only applying rotation and reflection to these units. Observe, that although Z-order has a flavour of self-similarity as one can image from Figure 1(c), it is a non-continuous data order in our sense. In the sequel we are only interested in continuous data orderings.

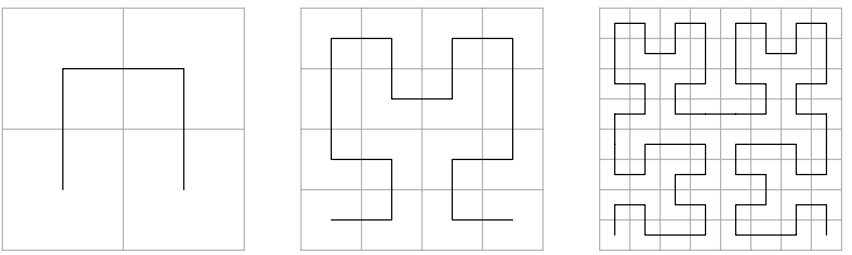


Fig. 3. First three stages of the Hilbert space-filling curve

3 Data Order and Its Generalization

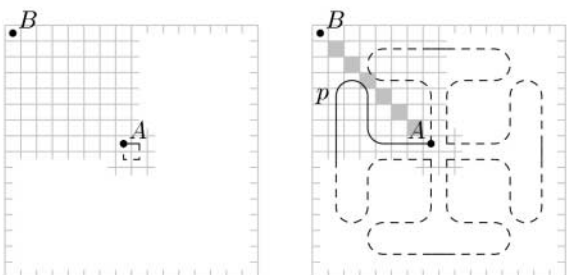
In this section we examine the question, whether there exists a data order which is closed under rotation and reflection. Up to now we only have seen one example with this property, namely the Hilbert order on a 2×2 grid—see Figure 3. It is clear, that Hilbert order on larger grids is not closed under rotation and reflection from the dihedral group due to the "sticky" ends of the data order. In general, it is easy to see that non-cyclic continuous data orderings can not be closed under rotation and reflection. The next theorem shows, that the situation is even worse, since for large enough grids even no cyclic data order with the property we are looking for exists.

Theorem 1. *Let n be a power of two greater than or equal to four. Then there is no cyclic data order $\varphi : \{1, \ldots, n^2\} \to \{1, \ldots, n\} \times \{1, \ldots, n\}$ which is closed under the dihedral group D_4.*

Proof. For the sake of a contradiction assume that a cyclic data order φ on an $n \times n$ grid which is closed under the dihedral group D_4 exists.

Divide the $n \times n$ grid in four equally sized sub-grids and consider cells A and B with coordinates $(\frac{n}{2}, \frac{n}{2} + 1)$ and $(1, n)$, respectively, i.e., the lower right and upper left cell, respectively, of the upper left sub-grid. Then we have to consider two cases. First, if A is connected to one of its neighbouring sub-grids, then we immediately obtain a contradiction. This is due to the fact that all to cell A corresponding cells—by rotation and reflection—are connected, and build a cycle without entering all cells in the grid. This is not possible in a cyclic data order. The situation described is seen in Figure 4(a).

Next consider the case when A is connected to the interior of the leftmost upper sub-grid. Then starting the indexing process in cell A leads us to a path p



(a) 1st case. (b) 2nd case.

Fig. 4. An assumption on the connectivity of cells is drawn with a solid line, while connectivity induced by rotation and reflection is drawn with a dashed line

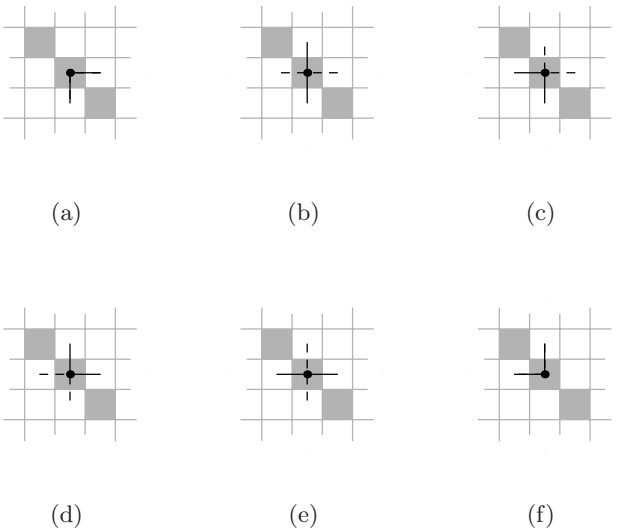


Fig. 5. All possible connections of a gray shaded cell. An assumption on the connectivity of cells is drawn with a solid line, while connectivity induced by rotation and reflection is drawn with a dashed line

through that sub-grid eventually entering another sub-grid. Without loss of generality we may assume that the first step starting in A leads to the west and that we may enter the leftmost lower sub-grid. This already induces a cyclic ordering, which must be consistent with φ. The situation described is depicted in Figure 4(b). If path p doesn't include the cells on the diagonal between A and B—these cells are shaded in Figure 4(b), then the induced ordering is *not* space-filling. Thus a contradiction. On the other hand, if at least one of the cells on the diagonal between A and B is part of the path p, then we have to consider six cases how the path goes through this cell. The six possibilities are shown in Figure 5. Obviously, cases (a) and (f) are not possible since this would contradict our assumption that p must enter another sub-grid. The remaining four cases (b)–(e) immediately lead to a contradiction since by rotation and reflection the cell under consideration gets degree four. Thus, this is also a contradiction, and therefore shows that no cyclic data order closed under the dihedral group D_4 exists. □

It is worth mentioning that the previous theorem can be strengthened such that n is any natural number greater than or equal to three. The theorem above easily generalizes to even numbers, while the missing odd case is seen as follows: In an $n \times n$ grid the interior cell A with coordinates $(\lceil \frac{n}{2} \rceil, \lceil \frac{n}{2} \rceil)$ must be connected to one of its neighbouring cells, and hence by rotation and reflection to all its four neighbours with Euclidean distance one. But then the cell under consideration

has degree four, which is a contradiction since every point $\varphi(i)$ has degree two in the graph $D(\varphi)$. Thus, we have shown the following corollary.

Corollary 1. *Let $n \geq 3$. Then there is no cyclic data order φ on an $n \times n$ grid which is closed under the dihedral group D_4.* □

The main argument in the previous proofs was that the closure under rotation and reflection implies that a grid graph must have vertices with degree four. Since this is not possible with an ordinary data order, we have obtained our contradiction. On the other hand this also shows that we must allow multiple access to grid point since a degree four vertex is traced twice. This immediately leads to generalized data orderings, which are defined as follows.

Let m and n be natural numbers. A *generalized data order* ψ on an $n \times n$ grid is a onto mapping

$$\psi : \{1, \ldots, n^2 + m\} \to \{1, \ldots, n\} \times \{1, \ldots, n\}.$$

As in case of ordinary data orderings we can associate a graph $D(\psi)$ with ψ. A generalized data order ψ is called *continuous* if and only if (1) $d_2(\psi(i), \varphi(i + 1)) = 1$ for all $1 \leq i < n^2 + m$, and (2) the associated graph $D(\psi)$ can be drawn in one stroke without hitting an edge twice. A continuous generalized data order φ is *cyclic* if and only if $d_2(\varphi(1), \varphi(n^2)) = 1$. By definition an ordinary (cyclic, continuous, respectively) data order is a (cyclic, continuous, respectively) generalized data order, but not necessarily *vice versa*. The next lemma shows that the graph $D(\psi)$ of a generalized data order fulfills the necessary condition to be closed under the dihedral group D_4. We omit the straight-forward proof of this lemma.

Lemma 1. *If ψ is a generalized data order, then every vertex in the graph $D(\psi)$ has even degree.* □

At this point it is still not clear whether a generalized data order satisfying our requirements exists. In the sequel we answer this question positively by constructing such a data order, which we call *Shamrock* order or for short *S*-order, due to its shape. Analogously to Hilbert order we obtain in an inductive manner a generalized data order for $n \times n$ grids, where n is a power of two. However, there is a slight decisive difference. Whereas in Hilbert order the basic building blocks are squares (cf. Figure 3), the construction of the *S*-order is easier to describe using triangles to which only rotation is applied.

We visualize the construction by means of an 0L type grammar, hence forcing the rewriting to take place simultaneously at every nonterminal. A similar approach to describe the construction of space-filling curves was undertaken by Asano *et al.* [2]. A nonterminal in the grammar represents an upper triangle of an $n \times n$ grid together with a line segment. The start and endpoints of a line segment are marked by a dot to control the rule application and for better orientation. Hence to obtain the entire area of an $n \times n$ grid one has to put two triangles in an up-side down manner together. The axiom is depicted in Figure 6(a). The rewriting process goes through a number of iteration, where each

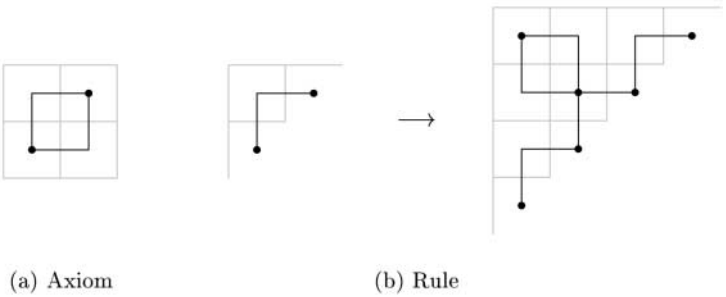


Fig. 6. Axiom and rule of the 0L type grammar to construct the S-order

iteration rewrites all non-overlapping triangles according to an application of a rule in our grammar as shown in Figure 6(b). Observe, that the 0L type grammar is allowed to rotate triangles. Figure 7 displays the first few iterations of the S-order starting with a single triangle instead of a whole square; the drawings are properly scaled such that each has same size. Moreover, in the first and second iteration we have indicated how to draw the line in one stroke without hitting an edge twice.

Hence we have shown the following theorem.

Theorem 2. *Let n be a power of two. Then there is a cyclic generalized data order $\psi : \{1, \ldots, n^2 + m\} \to \{1, \ldots, n\} \times \{1, \ldots, n\}$ which is closed under the dihedral group D_4.* □

It remains to compute the number m for S-order since this is the important parameter that will influence the compression ratio of RLE. Let n be a power of two for some constant $k \geq 1$. Then let E_k denote the number of degree-two points that hit the diagonal of a triangle that fits on an $n \times n$ grid and I_k be the number of intersecting points in a triangle that fits on an $n \times n$ grid. From Figure 7 one obtains that the first few values of (E_k, I_k) are $(0, 0)$, $(2, 1)$, $(6, 7)$, etc. In general $2 \cdot I_k + E_k$, which equals the number of intersecting points of the whole grid of appropriate size, obeys the recurrence

$$2 \cdot I_1 + E_1 = 0,$$

and

$$\begin{aligned} 2 \cdot I_k + E_k &= 8 \cdot I_{k-1} + 4 \cdot E_{k-1} + 4 \\ &= 4 \cdot (2 \cdot I_{k-1} + E_{k-1}) + 4, \end{aligned}$$

for $k \geq 2$, which has the solution

$$2 \cdot I_k + E_k = \frac{4^k - 4}{3}.$$

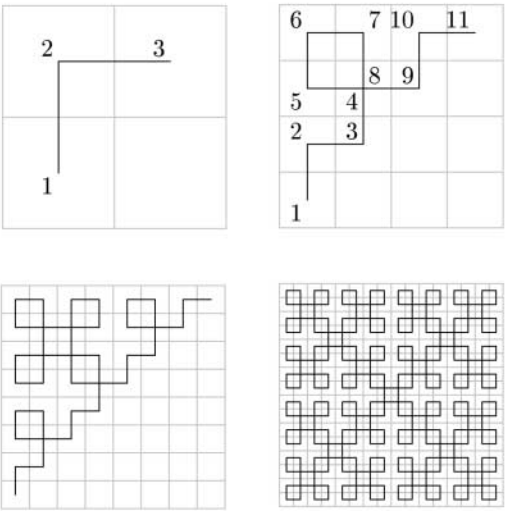


Fig. 7. The upper left triangle part of the first three stages of the S-order and its final shape

Comparing this number with the overall number of pixels on the corresponding $n \times n$ grid results in an surplus of

$$m = \frac{n^2 - 4}{3}$$

visited points. Thus, roughly a third of the points is used to obtain our closure under the dihedral group D_4.

At this point the question arises, whether the constructed order is best possible with respect to m, i.e., the number of points visited more than once. In fact, for generalized data orderings constructed in a recursive manner by an 0L type grammar with nonterminals in triangle form, we can show that one cannot do better than $m = \frac{n^2-4}{3}$. This is seen as follows. Consider the triangle of a right-hand side of a rule. Partition this triangle counter clock-wise into a smaller triangle, a square, and a smaller triangle again. These three substructures have to be connected by edges in order to obtain a generalized data order. Moreover, the endpoint of these edges under consideration have degree four by Lemma 1. Degree two is not possible, because the drawing in the square area without outside connecting edges must be already closed under the dihedral group D_4 and thus has no vertices with degree one. Then the recurrence that determines m reads as

$$2 \cdot I_k + E_k = 4 \cdot (2 \cdot I_{k-1} + E_{k-1}) + c,$$

for $k \geq 2$ and some constant c. Here c is the overall number of vertices that are visited twice in a complete square built by two right-hand sides of the rule under consideration. By our argumentation above, c must be at least four, because one may place the connecting edges of the substructures near the midpoint of the hypotenuse, which results in four additional points (instead of eight). This shows that the S-order is best possible with respect to m.

4 Image Operations and S-order

We discuss how to implement certain image operations under the assumption that RLE in combination with S-order is used to store raster images. It is seen that a certain variant of a double-queue automaton is already suitable to perform rotation, reflection, zooming, and inversion on the raster image only using its encoded form. Due to the lack of space we only briefly describe how to perform rotation and reflection.

First consider a 4×4 grid as shown in Figure 7. Then one easily observes that a rotation by $\frac{\pi}{2}$ changes the numbering of the appropriate S-order equal to the permutation

$$d = \begin{pmatrix} 1\ 2 \ldots 15\ 16 \ldots 20 \\ 6\ 7 \ldots 20\ \ 1\ \ \ldots\ \ 5 \end{pmatrix}$$

on 20 points. This nicely generalizes to

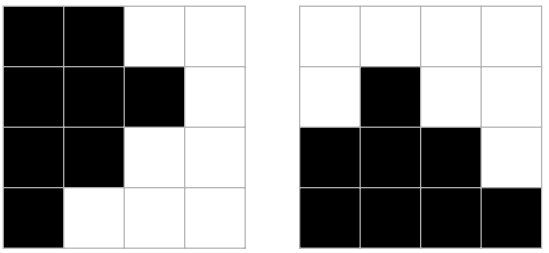
$$d = \begin{pmatrix} 1 & 2 & \ldots\ n^2-1\ n^2 & \ldots\ \frac{4n^2-4}{3} \\ \frac{n^2-1}{3}+1 & \frac{n^2-1}{3}+2 & \ldots\ \frac{4n^2-4}{3}\ \ 1 & \ldots\ \frac{n^2-1}{3} \end{pmatrix}$$

for $n \times n$ grids, which can be verified by induction on n. Analogously, we find that the reflection s on $n \times n$ grids is mimicked by the permutation

$$s = \begin{pmatrix} 1 & 2 & \ldots\ n^2\ n^2+1 & \ldots\ \frac{4n^2-4}{3} \\ n^2\ n^2-1 & \ldots\ 1 & \frac{4n^4-4}{3} & \ldots\ n^2+1 \end{pmatrix}.$$

¿From the formal language point of view, the rotation corresponds in a certain sense to an operation that maps a word uv to a word vu, while a reflection maps uv to $u^R v^R$, where w^R denotes the mirror image of w. Based on this easy observations we develop a pseudo code for a variant of a double-queue automaton that operates on the RLE of the raster image. To this end we have to make some assumptions on the RLE.

Without loss of generality we assume that the encoded data consist of a head and a body section, where the head contains information on the raster image size and the body the actual RLE of the image using S-order. The body is a sequence of tuples of the form (■ : i) or (□ : i) indicating that a data stream with i repeated values of ■ or □, respectively, was seen during the S-order like traverse of the raster image. Consider the raster image shown in Figure 8(a). A S-order traverse, partially depicted in the second drawing of Figure 7, reads out the pixel colors and results in the 20 element color sequence



(a) Original. (b) Rotated.

Fig. 8. An example of a 4×4 monochrome raster image

■■■■■■■■■□□□■□□□□□■□, which is simply encoded as (■ : 9)(□ : 3)(■ : 1)(□ : 5)(■ : 1)(□ : 1). The RLE of the rotated raster image as shown in Figure 8(b) equals the tuple sequence (■ : 4)(□ : 3)(■ : 1)(□ : 5)(■ : 1)(□ : 1)(■ : 5).

Now we are ready to give the pseudo code for a one-way double-queue automaton with additional registers to perform simple arithmetic tasks such as addition and multiplication of integers. The double-queue is manipulated by the functions first, rest, last, least, prefix, and postfix, which have the intuitive meanings. The isempty function checks whether the double-queue is empty or not–for further information on these operations we refer to Bauer and Goos [3]. Operation next acts on the input. It gives back a tuple of the form (■ : i) or (□ : i) and moves the input head to the right. The below given pseudo-code algorithm for the raster image rotation is based on our previous observations and uses the implicitly defined self-explaining and pre-initialized variables *dqueue* and *input*. It reads as follows:

```
1                                              initialization and simple calculations
2  position := 0; surplus := 0;
3  cutpoint := (n^2−1)/3;
5                                              search for the cut-point (n^2−1)/3
6  while position ≤ cutpoint do
7        element := next(input);
8        dqueue := postfix(dqueue, element);
9        position := position + element.number od;
11                                             remove last stored item
12 dqueue := least(dqueue);
14                                             store remaining part
15 surplus := position − cutpoint;
16 dqueue := postfix(dqueue,
18   (element.color : element.number − surplus));
```

```
21                                                    write overhanging pixels
22 write (element.color : surplus);
24                                                    write remaining input
25 while ¬iseof(input) do
26       write next(input) od;
28                                                    write previously stored part
29 while ¬isempty(dqueue) do
30       write first(dqueue) od;
```

The reader may have notice, that the algorithm is not optimal, since it doesn't verify whether it is possible to glue tuples with same color together after the cutting and recombination. For instance, rotating the raster image shown in Figure 8 twice, should result in the RLE (□ : 3)(■ : 1)(□ : 5)(■ : 1)(□ : 1)(■ : 9)(□ : 1) but our algorithm applied twice to the RLE of the original raster image produces the sequence (□ : 3)(■ : 1)(□ : 5)(■ : 1)(□ : 1)(■ : 5)(■ : 4)(□ : 1) instead. This is not a serious problem and can be fixed with slight changes to the above given pseudo code.

Similar algorithms can be given for the other raster image operations mentioned at the beginning of this section. Only in case of zooming a slight enhancement of the double-queue automaton is necessary in order to be able to construct the S-order for larger or smaller sized images. We have to omit the details and refer to a longer version of this paper. In all cases the running times of the algorithms are linear in the input size.

5 Experiments

We have conducted experiments with data order, especially S-order, on monochrome raster images. First of all to experimentally verify that S-order can compete against already existing data orderings and secondly to implement image operations as discussed in the previous sections. A screenshot of the Java 1.3 program used to perform our test runs on Intel Celeron 400 MHz and AMD Athlon 1 GHz machines and some sample input is shown in Figure 9—unfortunately, the buttons on the screenshot are labelled in German. Both machines use Windows 98 as an operating system and have 256 MB RAM. The input to our program are files in bmp-format.

As test data sets we have chosen a wide range of images from standard examples like text fragments and artificial images as, e.g., chess boards and fractals, up to non-common ones as, e.g., silhouettes of islands inspired by some of our geographical references, and modern paintings. We compare the various data orderings paying special attention to S-order by their compression ratio, which is the size of the output data divided by the size of the input data. The results are summarized in Figures 10 and 11 and can be interpreted as follows.

Figure 10(a) shows the results for Hilbert order compared to the traditional row order and column order. As expected Hilbert order is slightly better than the others, since most of the dots are above the diagonal that splits the first quadrant into half. The results for Z-order are nearly completely the other way

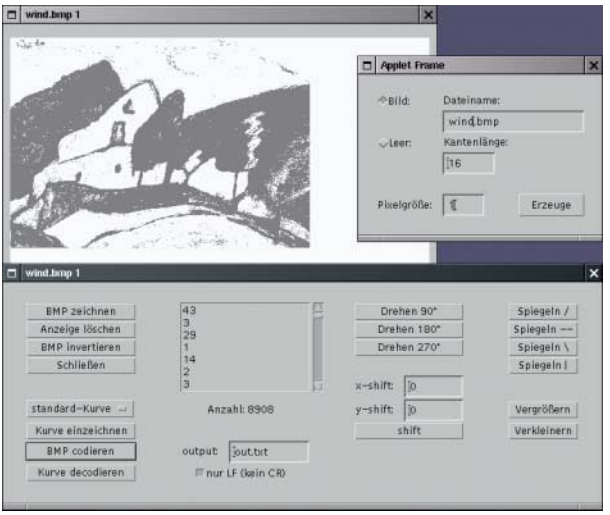


Fig. 9. Screenshot of the Java 1.3 program written to perform some experiments and a sample image—"Wind and Weather" (1910) of G. Münter

around as shown in Figure 10(b). There, maybe due to the non-local nature of Z-order, most of the dots are below the diagonal. Hence row and column order mostly supersedes Z-order. Finally in Figure 10(c) the results for S-order are shown. Surprisingly S-order is weaker than the traditional orderings, which comes from the surplus of roughly a third of pixels points. The gray shaded area in Figure 10(c) indicates the region of compression ratio induced by these additional points. Taking a closer look, one observes, that the dots cover nearly the same region as in case of Z-order and in fact the overall impression is like in the Hilbert order case, only slightly shifted to the lower end in the gray shaded region. The outliers at the left hand sides of the diagrams are generated by chess board-like images.

Nevertheless, the dominance of the space-filling Hilbert order and Z-order is not as significant as one might have expected from previous results in the literature. To confirm this intuition, we have to continue our experiments to clear this up.

Finally in Figures 11(a)–11(c) row order, column order, and S-order are compared to Hilbert order and Z-order. For the former two figures we come to the same conclusion as for Figures 10(a) and 10(b), respectively. It remains to consider Figure 11(c) with the results for S-order. As in Figure 10(c) the gray shaded region is induced by the additional points of the S-order traverse. One observes, that S-order is roughly comparable to Z-order, which nicely fits to the interpretation of Figure 10(b). Interestingly, most of the dots for the Hilbert order lie exactly on the lower end of the gray shaded region, which shows that Hilbert order and S-order are similar with respect to their compression ability and the difference in compression ratio is determined by the surplus of

pixels, only. This is exactly the result which one can await from the theoretical investigations in Section 3.

6 Conclusions

The results on the S-order nicely fit to already existing experimental investigations on data orderings and look very promising. We have seen that S-order is comparable to Z-order from the compression point of view, although the size of the domain of S-order is $\frac{4n^2-4}{3}$ instead of n^2 as in case of traditional data

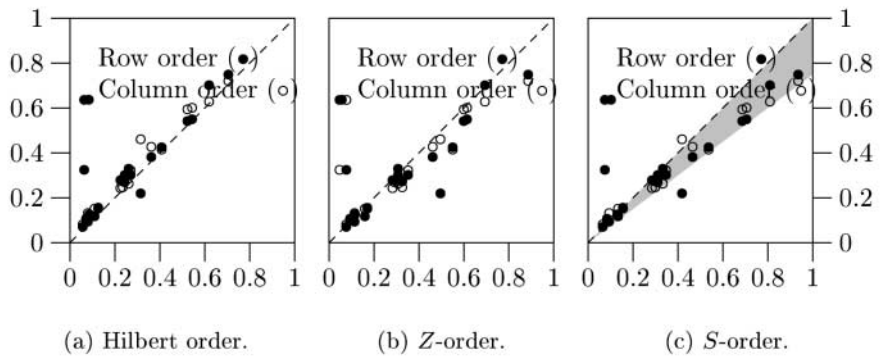


Fig. 10. Compression ratio of Hilbert order, Z-order, and S-order (x-axis) compared to row and column order (y-axis)

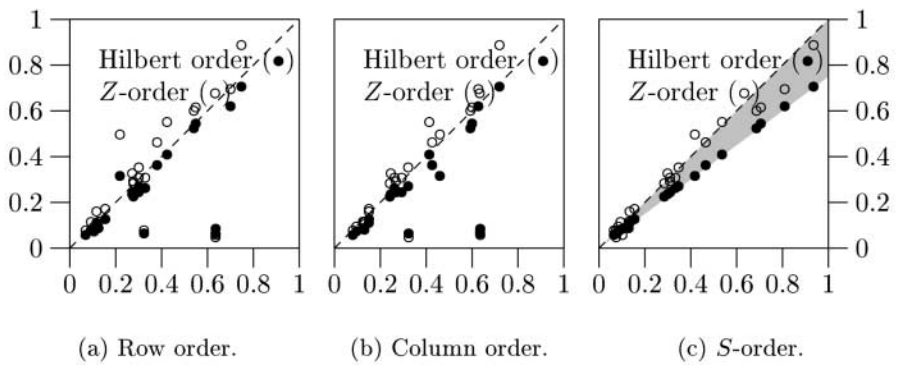


Fig. 11. Compression ratio of Row order, Column order, and S-order (x-axis) compared to Hilbert order and Z-order (y-axis)

orderings. Moreover, our experiments suggest that in most cases Hilbert order is not better than $\frac{3}{4}$ of S-order, and that in addition S-order has the advantage to support certain image operations like rotation, reflection, and zooming well, since it was especially designed for this purpose. To overcome the slight obstacle on compression ratio of S-order one might relax the condition on the closure under the dihedral group D_4, in order to get rid of twice visited pixel points, but still supporting image operations well. Future research will reveal to what applications and to what degree S-order or a relaxed variant can be advantageous.

Acknowledgments

We thank Björn Fay who has written the Java 1.3 program and conducted the experiments on various data orderings for us.

References

[1] D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990. 162

[2] T. Asano, D. Ranjan, Th. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, 1997. 168

[3] F. L. Bauer and G. Goos. *Informatik—Eine einführende Übersicht II.* Springer, 1984. 172

[4] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, Juni 1998. 162

[5] M. F. Goodchild and A. W. Grandfield. Optimizing raster storage: an examination of four alternatives. In *Proceedings of the 6th International Symposium on Automated Cartography*, pages 400–407, Ottawa, Ontario, Canada, 1983. 162

[6] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891. 165

[7] W. Kou. *Digital Image Compression: Algorithms and Standards.* Kluwer, 1995. 163

[8] G. M. Morton. A computer oriented geodetic data base, and a new technique in file squencing. Technical report, IBM, Ottawa, Ontario, Canada, 1966. 162

[9] D. Salomon. *Data Compression.* Springer, 1997. 161

[10] M. Weinstein. *Examples of groups.* Polygonal Publishing House, Passaic, New Jersey, 1977. 164

Enhancements of Partitioning Techniques for Image Compression Using Weighted Finite Automata

Frank Katritzke[1], Wolfgang Merzenich[1], and Michael Thomas[2]

[1] Department of Electr. Eng. and Computer Science, University of Siegen, Germany
[2] Mathematical Department, University of Siegen, Germany

Abstract. WFAs (weighted finite automata) are efficient structures for the storage of digital images. The choice of the image partitioning technique is important to achieve good compression results. In this paper we examine the fitness of various promising techniques by measuring the compression performance at well-known test images.

1 Introduction

The efficient storage and transmission of digital images is gaining increasing interest, for example in multimedia and WWW applications. Pictures require a huge amount of storage capacity, even in compressed form (as GIF or JPEG). For this reason, many researchers study the area of digital image compression extensively.

Weighted finite automata (WFAs) are structures which have successfully been utilized for efficient storage of digital images. One of the most important aspects for data compression with WFAs is the choice of the image partitioning technique. In this paper we examine the suitability of some techniques for our purposes. The most widely used partitioning techniques for WFAs are quadtrees and bintrees. We examine the application of HV partitioning which is a common technique for image partitioning in image coding with iterated function systems (IFS).

The reader already familiar with weighted finite automata may skip to section 4 on page 184.

2 Basic Techniques for Image Partitioning

The technique of WFA coding is based on partitioning an image into sub-images. Such an image partitioning was introduced in a scheme named quadtree image compression. It can be refined using bintrees and HV bintrees. In the following sections, we assume that the input image is a square matrix with a side length of 2^k, $k \in \mathbb{N}$. We will call elements of an image pixel.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 177–189, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2.1 Quadtrees

The original image partitioning in the WFA technique uses quadtrees [Cul93]. The *quadtree* is a hierarchical data structure based on the well known principle of recursive partitioning (divide and conquer principle). We examine the *region quadtree*, which divides the input image recursively into four quadrants of equal size. The quadtree is therefore a tree of degree four, with each inner node having four children. The leaves have to fulfill a given criterion of simplicity. In the original quadtree image compression algorithm, the input image is divided until all pixels in a given quadrant have equal gray values.

For an illustration of this procedure see Figure 1. In this drawing, the nodes will be traversed in the order represented by the tree on the right. The leaves store the gray values "B" (black) and "W" (white). The inner nodes are labeled with "G" (gray) to indicate that they have to be partitioned further.

A quadtree is suited for image compression because large areas can often be stored in a single node. The quadtree can also be used for lossy image coding if the recursive division is stopped if a special "homogeneity property" is fulfilled by the addressed segment. The associated node of the quadtree could store a gray value which minimizes a given distortion function. The average value is often used for this task. The quadtree structure may be stored by using four bits per node by storing for each inner node four booleans indicating whether the corresponding quadrants have to be split or not. By using entropy coding the quadtree structure may be stored with significantly less than four bits per node.

2.2 Bintrees

An important variant of quadtrees are the *bintrees*, where at each node the image is subdivided only in two parts instead of four. Horizontal and vertical subdivisions are applied alternately in the niveau of the tree. If only even numbered leaf niveaus were permitted, the bintree division would be the same as that of quadtrees. Using bintrees the division process can be stopped earlier, so the associated parts of the image contain more pixels and can therefore be coded more efficiently. We observe that the bintree structure can be stored using two bits per node without entropy coding in the same way as explained for quadtrees. For an illustration of a bintree see Figures 2 and 6.

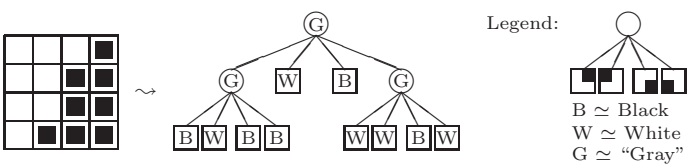


Fig. 1. Region quadtree. The ordering of the quadrants and the node labels are indicated in the legend on the right

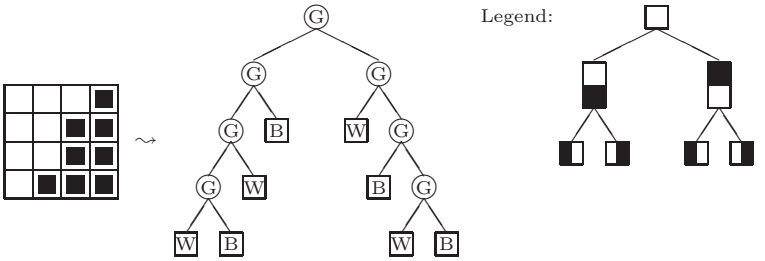


Fig. 2. Region bintree. The ordering of the segments and the node labels are indicated in the legend on the right

3 Weighted Finite Automata

Now we introduce the main topic of interest in this paper: the weighted finite automaton (WFA). Image coding with this structure was introduced by K. Culik and J. Kari [Cul93]. In order to simplify the approach, we introduce this kind of structure with quadtrees. As we see later, WFAs may be defined in a similar manner using bintrees and even HV bintrees (see section 4.1). Since a WFA may be stored with less bits than the original image, such automata are of interest for image compression.

3.1 Definition of a WFA

In order to examine images with methods of formal languages, we associate words over an alphabet Σ with 2-dimensional image segments. Let us first introduce basic notations of formal languages: let Σ be an alphabet, then the set Σ^n denotes all words (strings) of length n and $\Sigma^* := \bigcup_{n\geq 0} \Sigma^n$.

We call a function $f : \Sigma^* \to \mathbb{R}$ a *multi-resolution image*. We explain the concept of a weighted finite automaton for the quadtree case only and thus associate real numbers with the image segments defined by the nodes of a quadtree. These numbers will express the average gray value of the addressed image segment. In order to describe the quadtree nodes we label the four children of a node with elements of an alphabet $\Sigma = \{0, 1, 2, 3\}$ (and $\Sigma = \{0, 1\}$ in the non-quadtree cases considered in this paper). The string defined by concatenating the labels assigned to the quadrants on the path from the root to the desired node is associated with the image segment defined by that node. For an illustration of this addressing concept see Figure 3. To assign gray values to the image segments, we consider only multi-resolution images with the following property:

$$f(\sigma) = \frac{1}{|\Sigma|} \sum_{q\in\Sigma} f(\sigma q) \quad (\sigma \in \Sigma^*). \tag{1}$$

One can interpret such a function as a quadtree for $|\Sigma| = 4$ with infinite depth, where each node incorporates the average brightness of its children. A function which satisfies equation 1 is called *conservative* (or *average preserving*, [Cul93]).

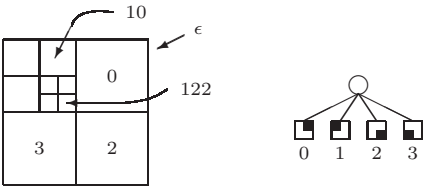


Fig. 3. Interpretation of a quadtree as a trie

Now we interpret the set of functions $f : \Sigma^* \to \mathbb{R}$ as a real vector space V with the following definitions of the required operations:

$$(f_0 + f_1)(\sigma) = f_0(\sigma) + f_1(\sigma) \quad (f_0, f_1 \in V, \sigma \in \Sigma^*), \tag{2}$$

$$(cf)(\sigma) = cf(\sigma) \quad (f \in V, c \in \mathbb{R}, \sigma \in \Sigma^*). \tag{3}$$

It is obvious that the set of conservative functions $\mathcal{D}$ is a linear subspace of the vector space V.

Definition 1. *A* weighted finite automaton *(WFA)* $\mathcal{A} = (S, \Sigma, W, I, F)$ *is specified by*

1. *a finite set of* states $S = \{s_0, \ldots, s_{|S|-1}\}$,
2. *a finite* alphabet $\Sigma = \{a_0, ..., a_{|\Sigma|-1}\}$,
3. *a* weight function $W : S \times \Sigma \times S \to \mathbb{R}$,
4. *an* initial distribution $I : S \to \mathbb{R}$,
5. *a* final distribution $F : S \to \mathbb{R}$.

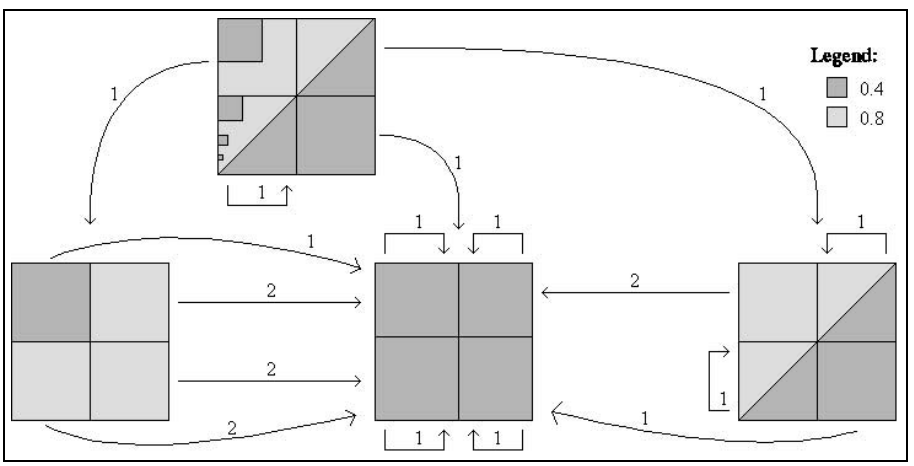


Fig. 4. Construction of a WFA. The edges are labeled with the corresponding weights and the labels of the edges are defined by the quadrants of the starting points

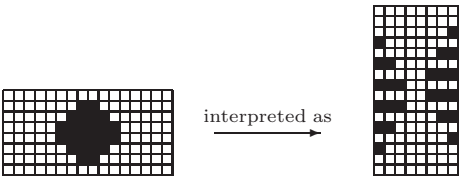


Fig. 5. Interpretation of a vector as an image of wrong resolution

As mentioned above, we either choose $\Sigma = \{0, 1, 2, 3\}$ (quadtree partitioning) or $\Sigma = \{0, 1\}$. If nothing else is stated, we will regard the quadtree case. We write $(W_a)_{i,j} = W(s_i, a, s_j)$ with $a \in \Sigma$, $s_i, s_j \in S$ and say that (s_i, a, s_j) is an *edge* of the weighted finite automaton iff $(W_a)_{i,j} \neq 0$. In order to perform calculations with the WFA $\mathcal{A}$, we will consider W_a as a real valued $|S| \times |S|$-matrix, I and F as $|S|$-dimensional (column) vectors.

We define for each state $s_i \in S$ a multi-resolution image Φ_i (*state image*) by

$$\Phi_i(a_0 \dots a_{k-1}) \ = e_i^t(W_{a_0} \dots W_{a_{k-1}} F) \tag{4}$$

where e_i is the i^{th} canonical unit vector.

We define the multi-resolution image given by the WFA $\mathcal{A} = (S, \Sigma, W, I, F)$ as a linear combination of the state images Φ_i

$$f_{\mathcal{A}} \ = \ \sum_{j=0}^{|S|-1} I_j \Phi_j, \tag{5}$$

or expressed in another way

$$f_{\mathcal{A}}(a_0 \dots a_{k-1}) = I^t(W_{a_0} \dots W_{a_{k-1}} F) \tag{6}$$

for $k \in \mathbb{N}$ and $a_0 \dots a_{k-1} \in \Sigma^*$.

Equation 6 defines a (slow) WFA decoding algorithm, since we can calculate the associated brightness for each pixel in the target image with this formula. Faster WFA decoding algorithms are given in [Cul93, Udu96].

3.2 The First WFA Coding Algorithm Introduced by Culik and Kari

In order to construct a WFA for a given image, we have to construct the vectors I, F and the matrices defined by W_a ($a \in \Sigma$). Since the vector I depends on the encoding algorithm, we will specify this vector later. From the definition of the state images (equation 4) we can infer by setting $k = 0$

$$\Phi_i(\epsilon) = e_i^t F = F_i \tag{7}$$

that F_i equals the average brightness of the state image Φ_i. We examine equation 4 further to see how the matrices W are constructed,

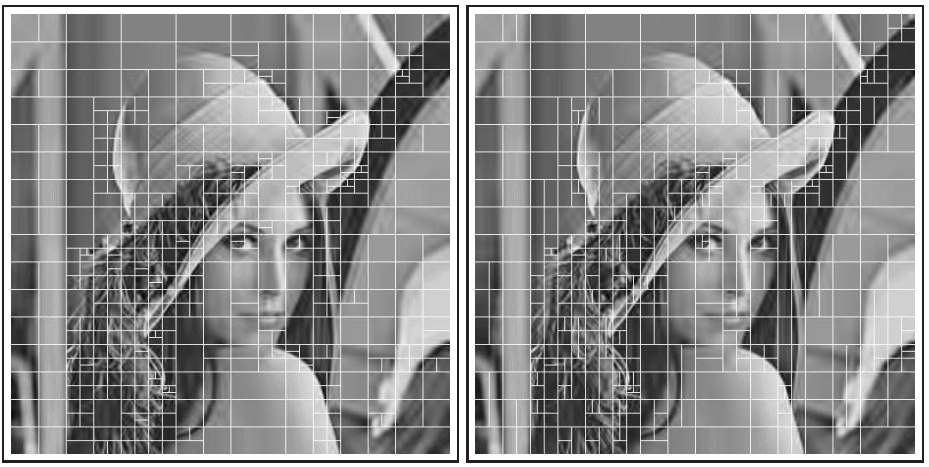


Fig. 6. Bintree (left) and light HV partitioning (right) of the image Lenna. Differences may be observed at the left bar and the right hair strand because of the strong vertical edges

$$\begin{aligned}
\Phi_i(a_0 a_1 \dots a_{k-1}) &= e_i^t W_{a_0} W_{a_1} \dots W_{a_{k-1}} F && (8)\\
&= \left(e_i^t W_{a_0}\right)\left(W_{a_1} \dots W_{a_{k-1}} F\right) && (9)\\
&= \left(e_i^t W_{a_0}\right)\begin{pmatrix} e_0^t W_{a_1} \dots W_{a_{k-1}} F \\ \vdots \\ e_{|S|-1}^t W_{a_1} \dots W_{a_{k-1}} F \end{pmatrix} && (10)\\
&= \left((W_{a_0})_{i,0}, \dots, (W_{a_0})_{i,|S|-1}\right)\begin{pmatrix} \Phi_0(a_1 \dots a_{k-1}) \\ \vdots \\ \Phi_{|S|-1}(a_1 \dots a_{k-1}) \end{pmatrix} && (11)\\
&= \sum_{j=0}^{|S|-1} (W_{a_0})_{i,j} \Phi_j(a_1 \dots a_{k-1}). && (12)
\end{aligned}$$

In this equation, we see that the quadrants of the state images are linear combinations of other state images. With these preliminaries, we can define a straightforward WFA construction algorithm.

In order to ease the next explanations, we introduce the following notation for conservative functions $\Psi \in \mathcal{D}$:

$$\Psi_{\sigma_0}(\sigma_1) = \Psi(\sigma_0\, \sigma_1) \quad (\sigma_0, \sigma_1 \in \Sigma^*) \tag{13}$$

meaning that we zoom Ψ into the image segment $\sigma_0 \in \Sigma^*$.

The method in the next listing constructs a WFA which approximates the conservative function Ψ. First a state representing the whole input image is

```
operateImage(Ψ)
  begin
  create the state s_0 ;
  Φ_0 := Ψ_ε ;
  n := 1;
  S := {s_0};
  U := {s_0};      {set of unprocessed states.}
  while U ≠ ∅
    begin
    choose a state s_i ∈ U ;
    for a ∈ Σ
      begin
      if ∃(r_0, ..., r_{n-1}) ∈ ℝ^n : (Φ_i)_a = r_0Φ_0 + ... + r_{n-1}Φ_{n-1}
        store the coefficient vector (r_0, ..., r_{n-1})
      else
        begin
        create a new state s_n ;
        S := S ∪ {s_n};
        U := U ∪ {s_n};
        Φ_n :=(Φ_i)_a ;
        n := n + 1;
        store the coefficient vector (0, ..., 0, 1)
        end
      end
    U := U \ {s_i}
    end
  I :=(1, 0, ..., 0)
  end;
```

Fig. 7. The first WFA coding algorithm

created. Afterwards the quadrants of a given state image are approximated by other state images. If that does not work, a new state is generated for the corresponding quadrant. It can be shown that the WFA generated by this method possesses the minimal number of states of all WFAs that generate Ψ [Cul93]. For more details concerning this method see [Cul95].

The coefficient vectors in the above algorithm may be interpreted as the rows of the weight matrices W_a for $a \in \Sigma$. So the coefficient vector belonging to the quadrant a in the state s_i is the i^{th} row of the matrix W_a.

Example 1. For an example of this coding process see Figure 4, where the image on the top is approximated. Assume that the quadrants are processed[1] in the order 1, 2, 0, 3. The generated automaton has four states and Figure 4 shows the state images of these states. The upper state image shows Ψ_ϵ (the original image), the three lower state images are (from left to right) Ψ_1, Ψ_2 and Ψ_0.

[1] Note that the order in which the quadrants are processed is arbitrary. We have chosen this processing order because of visualizing reasons.

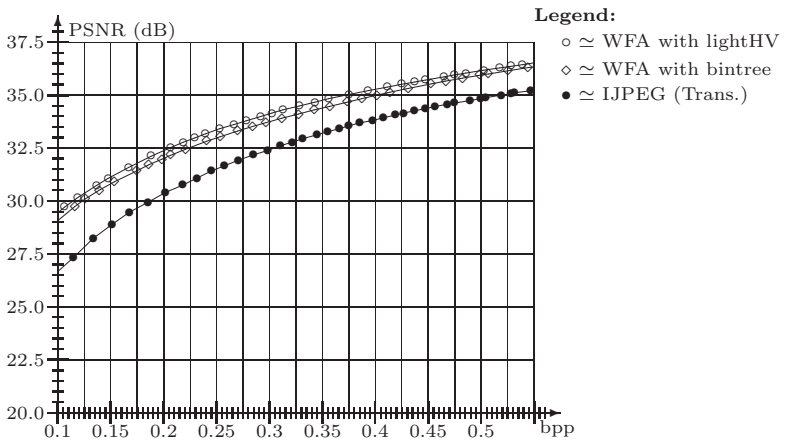


Fig. 8. Rate–distortion diagram of the image Lenna comparing the different techniques

The created WFA $\mathcal{A} = (S, \Sigma, W, I, F)$ has the following components:

- $S = \{s_0, s_1, s_2, s_3\}$, the state images of these states are shown in Figure 4 (top to bottom, left to right),
- $\Sigma = \{0, 1, 2, 3\}$ since in this example we apply quadtrees for image partitioning,
- $I^t = (1, 0, 0, 0)$ since the original image is the state image of the first state,
- $F^t = (0.5\overline{6}, 0.7, 0.4, 0.6)$ (these are the average brightnesses of the state images),
- and the following weight matrices:

$$W_0 = \begin{pmatrix} 0\,0\,0\,1 \\ 0\,0\,2\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \end{pmatrix}, W_1 = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,1\,0 \\ 0\,0\,2\,0 \end{pmatrix},$$

$$W_2 = \begin{pmatrix} 0\,0\,1\,0 \\ 0\,0\,2\,0 \\ 0\,0\,1\,0 \\ 0\,0\,1\,0 \end{pmatrix}, W_3 = \begin{pmatrix} 1\,0\,0\,0 \\ 0\,0\,2\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \end{pmatrix}.$$

4 Advanced Techniques for Image Partitioning

In section 2 we have introduced some basic techniques for image partitioning. Now we will examine how we can improve the WFA compressor using enhanced partitioning methods.

4.1 Light HV Partitioning

We can observe an important property of bintrees: if the first bintree partitioning is made horizontally, the image parts belonging to the leaves of the bintree are always either squares or twice as wide as high. Therefore, to give the bintree more flexibility, we have added an option for partitioning the square image segments either horizontally or vertically. We call this method for partitioning *light HV partitioning* since it is a special case of the HV partitioning method introduced in the next section. Note that this method requires the efficient storage of additional data and the improvement in image quality must be high enough to exceed the extra cost.

To explain some difficulties with the light HV partitioning, we express an image from now on as a $n \times m$ matrix with $n, m \in \mathbb{N}$ and not as a multiresolution image as in the last section. Such an image can be interpreted as a nm vector by using the raster scan order.

A problem of the light HV partitioning scheme is that different shapes of the image segments occur. We will explain this difficulty by using an example. Consider that an image segment of size 16×16 is to be partitioned. When the image segment is partitioned vertically, we would have to approximate two image segments of size 8×16. From the mathematical point of view, we could utilize all image segments of the sizes 8×16 and 16×8 and approximate the given image segment with these vectors. But in WFA coding this obvious technique yields bad results because of the following reason: if some image segments had the resolution 16×8 and are now interpreted as 8×16 image segments we would obtain such scattered images as can be seen in Figure 5 (right side). Such high frequencies arise because neighboring relations (and thus correlations) are destroyed.

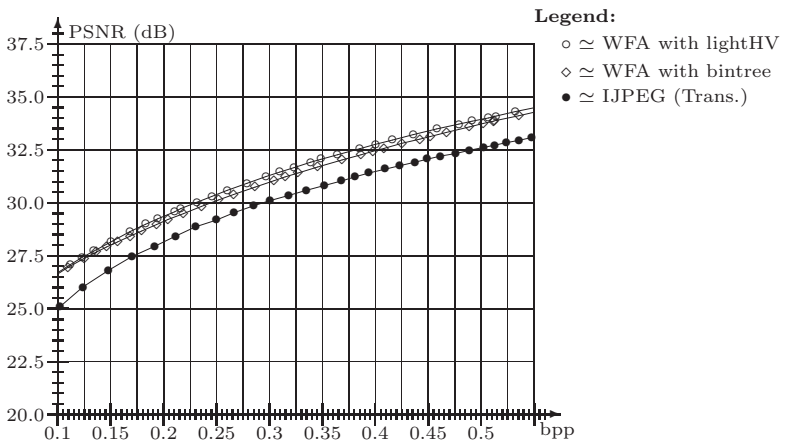


Fig. 9. Rate–distortion diagram of the image Boat comparing the different techniques

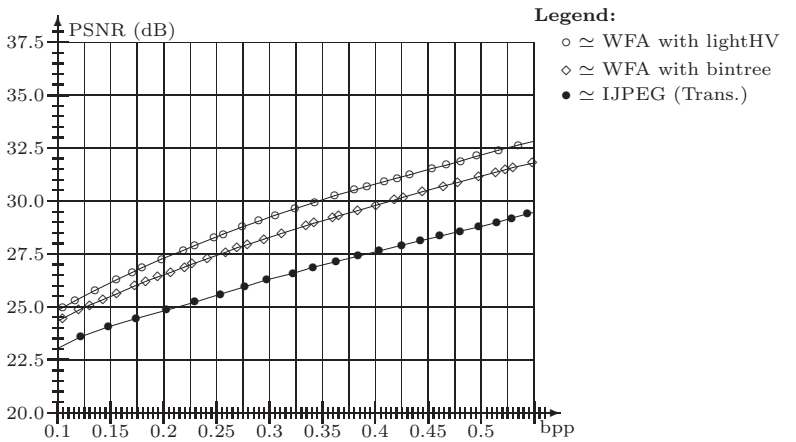


Fig. 10. Rate–distortion diagram of the image Barb comparing the different techniques

Another solution to this approximation problem would be to scale the images of resolution 16×8 up by the factor two in y-direction and afterwards scale the image down by the factor two in x-direction. This procedure wastes details in the image segment and can in the worst case lead to pixelization of the image segment and therefore to bad results. Another possibility would be to scale such image segments up to the next square image and approximate on that resolution. This procedure would both require more memory and twice the time to calculate the required scalar products. Our solution for this problem is the transposition of the image matrix. This technique avoids all drawbacks mentioned above and leads to good results.

For the determination of the partitioning line, we use the heuristic of Fisher which is described in the next section. For an illustration of this partioning scheme see Figure 6.

4.2 HV Partitioning

A more flexible approach for partitioning an image was introduced by Y. Fisher and S. Menlove in [Fis95b] with good experimental results using an IFS encoder. In this partitioning method, the codec is able to move the partitioning line from the middle towards outer regions of the image segment.

This approach was also tested in our implementation of the WFA codec. However, we observed that the gain in image quality did not compensate the extra cost for storing the information added to the bintree structure. For this reason, the light HV partitioning mentioned above was introduced, where extra storage cost is small. Another reason why this scheme does not yield good results in our implementation could be the problem of scaling. As mentioned in the last

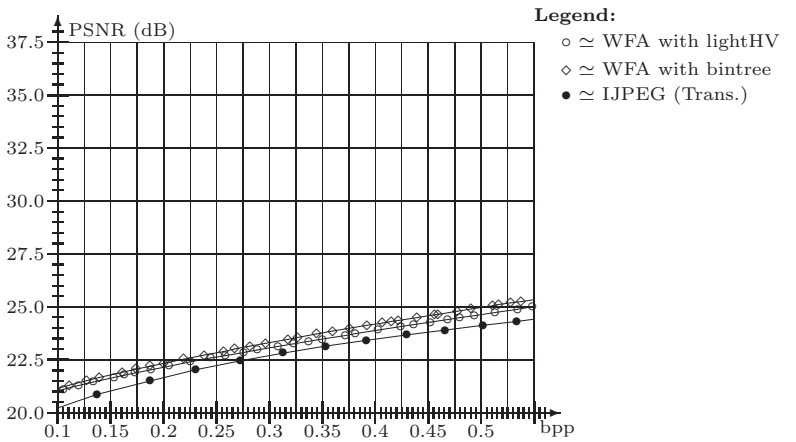


Fig. 11. Rate–distortion diagram of the image Mandrill comparing the different techniques

paragraph, image details are lost by scaling down. On the other hand, scaling up is not feasible.

We implemented the heuristic used in [Fis95b] to determine the partitioning line. For an $m \times n$ image segment with gray values $r_{i,j}$ ($i \in \{0, \ldots, n-1\}$, $j \in \{0, \ldots, m-1\}$), Fisher computes the *biased differences*

$$h_j = \frac{\min(j, m-j-1)}{m-1} \left| \sum_{i=0}^{n-1} (r_{i,j} - r_{i,j+1}) \right| \tag{14}$$

and

$$v_i = \frac{\min(i, n-i-1)}{n-1} \left| \sum_{j=0}^{m-1} (r_{i,j} - r_{i+1,j}) \right|, \tag{15}$$

chooses the biggest value of the set $\{h_0, \ldots, h_{m-2}, v_0, \ldots, v_{n-2}\}$ and splits the image segment at the corresponding index (horizontally if the biggest value is one of the h_i and vertically otherwise). The partitioning obtained is said to cut the image segments at strong horizontal and vertical edges while avoiding narrow image segments.

A better solution would be backtracking to determine the best partitioning line by checking which choice leads to the best results. That alternative has not been implemented because the running time would be too high for a practical solution.

However, we currently can not rule out the possibility that HV partitioning can be refined in another way to cooperate with WFA encoding. This point could be an interesting research topic for the future of WFA encoding.

Fig. 12. Gray scale images of the Waterloo image site. The images are called (from left to right): Barb, Boat, Lenna and Mandrill

Fig. 13. Decoded Images of Lenna. Left: bintree partitioning, 30.65 dB PSNR, 0.144 bpp, right: light HV partitioning 30.83 dB PSNR, 0.139 bpp.

5 Results

As you can see in Figures 8–11, the image quality is enhanced by approximately 0.25 dB PSNR (peak signal to noise ratio, see [Fis95]) at the same data rate by using the light HV partitioning scheme. For comparison we have added samples of the widespread image coding system JPEG [IJG]. Note that we have not added samples for the quadtree and HV partitioning scheme. As argued below, the bintree partitioning scheme is a generalization of the quadtree partitioning scheme and is thus at least as powerful. The HV partitioning scheme showed bad compression performance and high memory consumption in an early stage of implementation. Thus we did not optimize our implementation of WFA compressor to this partitioning method. Note that we have used the same ranges for all diagrams to visualize the compressibility of the different test images.

References

[Cul93] K. Culik II and J. Kari, *Image Compression Using Weighted Finite Automata.* Computers and Graphics **17**, 3 (1993) 305–313. 178, 179, 181, 183

[Cul95] K. Culik II and J. Kari, *Inference Algorithms for WFA and Image Compression.* Chapter of [Fis95] (1995) 243–258. 183

[Fis95] Y. Fisher (ed.), *Fractal Image Compression.* Springer-Verlag Berlin, Heidelberg, New York (1995). 188, 189

[Fis95b] Y. Fisher, *Fractal Encoding with HV Partitions.* Chapter of [Fis95] (1995) 119–136. 186, 187

[IJG] The Independent JPEG Group, *JPEG software release 6b.* `ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz` (1998). 188

[Ros79] A. Rosenfeld and L. S. Davis, *Image Segmentation and Image Models.* Proceedings of the IEEE, **67**, 5 (1979) 764–772.

[Udu96] A. Rao, V. D. Pandit and R. U. Udupa, *Efficient Decoding Algorithms for Weighed Finite Automata.* Department of Computer Science, S. J. College of Engineering, Myosore (1996). 181

[Wat] University of Waterloo, Ontario, Canada, *Waterloo BragZone: Comparison of Image Compression Systems.* `http://www.uwaterloo.ca/bragzone.base.html`. 189

A Utilized Test Images

One of the most popular test images is Lenna shown in Figure 12. Several variants of this image are in usage, thus making it hard to compare our results with others. In order to make the choice of images clear, we have used the widespread images of the Waterloo Fractal Image Coding Project [Wat].

B Some Decoded Images

For some decoded images with bintree and light HV partitioning see Figure 13.

Extraction of ε-Cycles from 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. Much attention has been brought to determinization and ε-removal in previous work. This article describes an algorithm for extracting all ε-cycles, which are a special type of non-determinism, from an arbitrary finite-state transducer (FST). The algorithm factorizes (decomposes) the FST, T, into two FSTs, T_1 and T_2, such that T_1 contains no ε-cycles and T_2 contains all ε-cycles of T. Since ε-cycles are an obstacle for some algorithms such as the factorization of ambiguous FSTs, the proposed approach allows us to by-pass this problem. ε-Cycles can be extracted before and re-inserted (by composition) after such algorithms.

1 Introduction

Much attention has been brought to the problem of non-determinism. There has been work on both determinization in general and ε-removal [1, 6, 5, among many others].

This article describes an algorithm for extracting all ε-cycles, which represent a special type of non-determinism consisting of consecutive arcs with the empty string ε as input label, from an arbitrary finite-state transducer (FST). The algorithm factorizes (decomposes) the FST, T, into two FSTs, T_1 and T_2, such that T_1 contains no ε-cycles and T_2 contains all ε-cycles of T. Jointly in a cascade, T_1 and T_2 describe the same relation and perform the same mapping as T.

Motivation: Some algorithms, such as the factorization of ambiguous FSTs [8, 7, 4], can only be performed on *real-time FSTs*, where every arc has exactly one symbol on the input side. Arcs with ε as input label are an obstacle for such algorithms. In many cases, an FST can be made real-time by removing its ε-arcs and concatenating their output labels with the output of adjacent non-ε-arcs. This classical method, however, is not applicable to FSTs with ε-cycles. To by-pass the problem, the ε-cycles of an FST, T, can be extracted by the approach below, where T is factorized into T_1 and T_2. Then, the ε-cycle-free and (at most) finitely ambiguous T_1 can be made real-time and further factorized into a sequential $T_{1,1}$ and an ambiguous flower transducer $T_{1,2}$ that contains no failing paths for any output string of $T_{1,1}$ [4]. Finally, the ε-cycles can be re-inserted by composing $T_{1,2}$ with T_2.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 190–201, 2002.
© Springer-Verlag Berlin Heidelberg 2002

1.1 Conventions

Input and Output Side: Although FSTs are inherently bidirectional, they are often intended to be used in a given direction. The proposed algorithm is performed relative to the direction of application. In this article, the two sides (or tapes or levels) of an FST are referred to as *input side* and *output side*.

Examples of Finite-State Networks: Every example is shown in one or more figures. The first figure usually shows the original network. Possible following figures show modified forms of the same example. For example, Example 1 is shown in Figure 1 to Figure 3.

Finite-State Graphs: Every FST has one initial state, labeled with number 0, and one or more final states marked by double circles. The initial state can also be final. All other state numbers and all arc numbers have no meaning for the FST but are just used to reference a state or an arc. 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 symbol pair `a:b`, `a` is the input and `b` the output symbol. Simple, i.e., unpaired labels represent identity pairs. For example, `a` means `a:a`.

Composition: In $T_1 \diamondsuit T_2 \diamondsuit T_3 = T_3 \circ T_2 \circ T_1$, the FST T_1 is applied first and T_3 last [2]. We will use the $\diamondsuit$-operator because we prefer left-to-right notation (and application) in general, and find it clearer in examples such as $(\texttt{a:b}) \diamondsuit (\texttt{b:c}) \diamondsuit (\texttt{c:d}) = (\texttt{a:d})$, compared to $(\texttt{c:d}) \circ (\texttt{b:c}) \circ (\texttt{a:b}) = (\texttt{a:d})$.

Special Symbols: The "`?`" denotes any symbol (except ε or $\hat{\varepsilon}$) when it is used in a regular expression. Both ε and $\hat{\varepsilon}$ mean the empty string and have the same effect when the FST is applied to an input sequence, but $\hat{\varepsilon}$ should be preserved in minimization and determinization. Greek letters are used to denote auxiliary symbols. Those have a "special" meaning and are distinct from the ordinary input and output symbols.

1.2 Preliminaries

An FST can be described by the six-tuple $T = \langle \Sigma, \Delta, Q, i, F, E \rangle$ with an input alphabet Σ, an output alphabet Δ, a state set Q, an initial state $i \in Q$, a set of final states $F \subseteq Q$, and a set of transitions E.

Given a transition $e \in E$, we denote its input label by $i(e)$, its output label by $o(e)$, its source state by $p(e)$, and its destination state by $n(e)$. The transition can be described by the quadruple $e = \langle p(e), i(e), o(e), n(e) \rangle$. Given a state $q \in Q$, we denote the set of its outgoing transitions by $E(q)$ and the set of its incoming transitions by $E^R(q)$. A path $\pi = e_1 \cdots e_k$ is an element of E^* with consecutive

transitions. To express that a transition e is on a path π, we write $e \,\overline{\overline{\in}}\, \pi$. To refer to a particular path in a figure, we give the arc numbers in ceiling brackets; e.g., $\pi = \lceil 100, 101, 102, 103 \rceil$ is a path consisting of the four named arcs. We denote by $P(q, q')$ the set of all paths $\pi_i(q, q')$ from q to q', by $C(q)$ the set of all cycles on q (i.e., all paths from q to q), and by $C_\varepsilon(q)$ the set of all ε-cycles on q , i.e., those cycles consisting only of arcs with ε as input label:

$$P(q, q') = \bigcup_i \{\pi_i(q, q')\} \tag{1}$$

$$C(q) = P(q, q) \tag{2}$$

$$C_\varepsilon(q) = \{\pi \in C(q) \mid \forall e \,\overline{\overline{\in}}\, \pi,\ i(e) = \varepsilon\} \tag{3}$$

We are particularly interested in simple ε-cycles $\widehat{C}_\varepsilon(q)$ on a state q which do not traverse any state more than once:

$$\widehat{C}_\varepsilon(q) \subseteq C_\varepsilon(q) \tag{4}$$

$$\widehat{C}_\varepsilon(q) = \{\pi \in C_\varepsilon(q) \mid \forall e, e' \,\overline{\overline{\in}}\, \pi,\ e \neq e' \Rightarrow n(e) \neq n(e')\} \tag{5}$$

We extend the notion of input and output labels to paths and sets of paths, cycles, or ε-cycles, and denote their sequences of input and output labels by $i(\pi(q, q'))$, $o(\pi(q, q'))$, $i(C_\varepsilon(q))$, $o(C_\varepsilon(q))$, etc. Note that $i(\,)$, $o(\,)$, and their arguments can be single elements or sets.

2 Basic Idea

Any arbitrary FST, T, containing ε-cycles can be factorized (decomposed) into two FSTs, T_1 and T_2, such that T_1 contains no ε-cycles and is therefore at most finitely ambiguous, and T_2 contains all ε-cycles of T. The set of ε-cycles $C_\varepsilon(q_i)$ of every state q_i in T is represented by a single arc mapping ε to an auxiliary symbol ξ_i in T_1. Instead of (perhaps infinitely) traversing $C_\varepsilon(q_i)$, ξ_i is emitted. All ξ_i are then mapped to the corresponding original $C_\varepsilon(q_i)$ in T_2 :

$$C_\varepsilon(q_i) \quad \longrightarrow \quad (\varepsilon : \xi_i) \quad \diamondsuit \quad (\xi_i : o(C_\varepsilon(q_i))) \tag{6}$$

Figure 1 shows a simple example of an FST with two ε-cycles, $C_\varepsilon(1) = \{\lceil 101, 102 \rceil\}$ and $C_\varepsilon(3) = \{\lceil 104, 105 \rceil\}$. The FST maps the input string `abc` to the output string `xyz`, and inserts an arbitrary number of substrings `rs` inside.

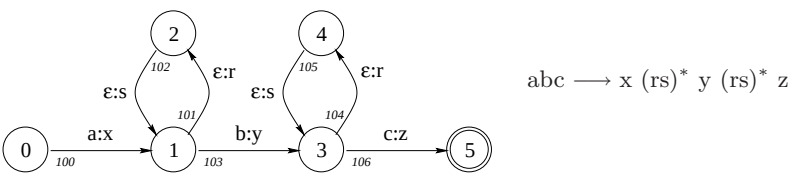


Fig. 1. Transducer T with ε-cycles (Example 1)

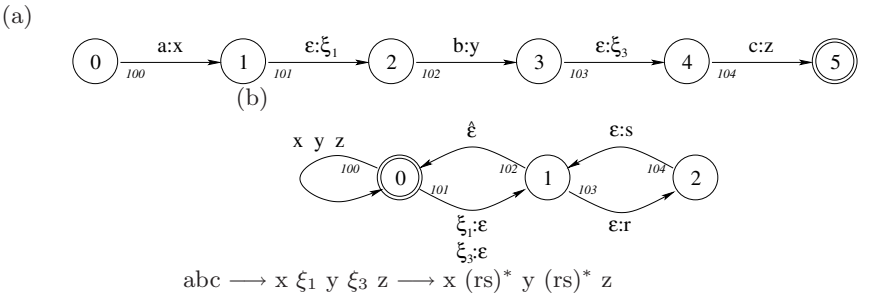


Fig. 2. Factorization of T into (a) an ε-cycle-free T_1 that emits auxiliary symbols, and (b) a T_2 that maps auxiliary symbols to ε-cycles (Example 1)

Figure 2 shows the same example after the extraction of ε-cycles (factorization). T_1 maps the input string `abc` to the intermediate string $\mathtt{x}\xi_1\mathtt{y}\xi_3\mathtt{z}$ (Fig. 2a). T_2 maps the auxiliary symbols, ξ_1 and ξ_3, to ε-cycles, and every other symbol of the intermediate string to itself (Fig. 2b). Although the auxiliary symbols are single symbols, they describe (sets of) ε-cycles. Since actually ξ_1 and ξ_3 describe equal ε-cycles in this example, it would be sufficient to use two occurrences of the same auxiliary symbol, e.g. ξ_1, instead. In such cases, the number of auxiliary symbols can be reduced a posteriori [3]. The $\hat{\varepsilon}$ denotes the empty string, like ε, but it should be preserved in minimization and determinization. Otherwise T_2 would become larger (Example 1 Fig. 2b, and Example 2 Fig. 9b).

T_1 can be converted into a real-time FST, without ε-arcs, by removing the ε-arcs and concatenating their output symbols with the output of adjacent non-ε-arcs.

An alternative to factorizing T into T_1 and T_2 would be representing T by a single FST, $\widehat{T}$, that is similar to T_1 but with more complex output labels that directly describe sets of ε-cycles $C_\varepsilon(q_i)$. Every $C_\varepsilon(q_i)$ in T would be reduced to a single ε-arc in $\widehat{T}$ (Fig. 3a). $\widehat{T}$ can be further converted into a real-time FST, without ε-arcs (Fig. 3b). This representation of ε-cycles is similarly to what can be seen, e.g., in [7, p. 221, Fig. 6], and is equivalent to our representation by two FSTs. In both cases one needs an algorithm (possibly very similar) for

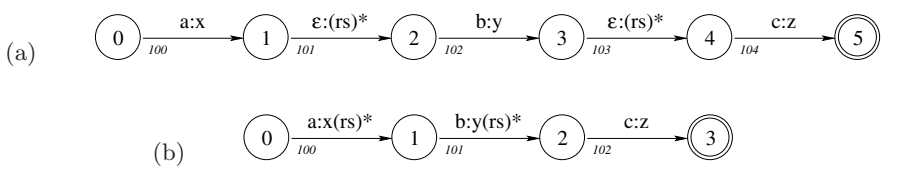
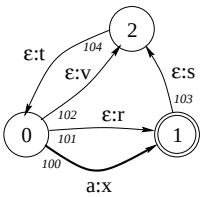


Fig. 3. Alternative representation of ε-cycles by complex labels (a) with ε-arcs or (b) as a real-time transducer without ε-arcs (Example 1)



$$\varepsilon \longrightarrow (\text{rst}|\text{vt})^* \text{ r}$$

$$\mathbf{a}\ \mathbf{a}^n \longrightarrow (\text{rst}|\text{vt})^*\ \mathbf{x}\ (\text{st}\ (\text{vt})^*\ \text{r})^* \left(\ \text{s}\ (\text{tv}|\text{trs})^*\ \text{t}\ \mathbf{x}\ (\text{st}\ (\text{vt})^*\ \text{r})^*\ \right)^n$$

Fig. 4. Transducer T with ε-cycles (Example 2)

identifying the $C_\varepsilon(q)$, extracting them from T, and constructing one or the other representation.

3 Algorithm

The above Example 1 contains only ε-cycles that could be removed by physically removing their arcs (Fig. 1). However, ε-cycles can be more complex. They can overlap with each other, with non-ε-cycles, or with other (non-cyclic) paths. This means, ε-cycles must be removed without physically removing their arcs.

Figure 4 shows a more complex example.[1] None of the ε-arcs *101*, *103*, and *104* can be physically removed because they are not only part of ε-cycles but among others also of the complete paths ⌈*101*⌉ and ⌈*100, 103, 104, 100*⌉ that accept the input strings ε and `aa` respectively.

3.1 Preparation

To extract all ε-cycles of an arbitrary FST, T, the algorithm proceeds as follows. First, T is concatenated on both ends with boundary symbols, # (Fig. 5). This operation causes that the properties of initiality and finality, so far only described by states, are now also described by arcs and can therefore be ignored by the algorithm (cf. all pseudo code).

Each state q_i in T is then assigned both information about its $\widehat{C}_\varepsilon(q_i)$ and an auxiliary symbol ξ_i that (at this stage) is considered as equivalent to $\widehat{C}_\varepsilon(q_i)$. The resulting FST is called T' (Fig. 5). For example, state 1 is assigned the set $\widehat{C}_\varepsilon(1) = \{\lceil 102, 105, 106 \rceil, \lceil 103, 106 \rceil\}$ and the auxiliary symbol ξ_1 which means that two ε-cycles consisting of the named arcs start at state 1 and are equivalent to ξ_1. These two ε-cycles generate the output substrings `(rst)`* and `(vt)`* respectively.

[1] In all figures of Example 2, thin arcs are used for ε-transitions and thick arcs for non-ε-transitions.

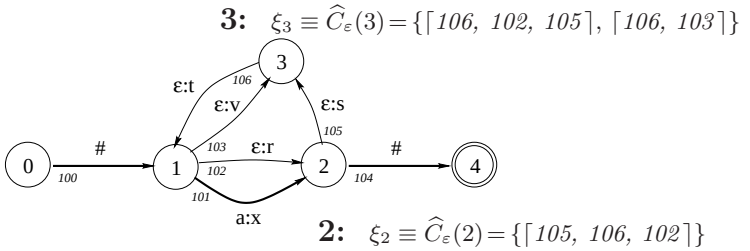


Fig. 5. Transducer T' with boundaries, auxiliary symbols, and ε-cycle information (Example 2)

There are different ways to compute the $\widehat{C}_\varepsilon(q)$ of all q. For example, starting from a state q, we traverse every ε-path that does not encounter any state, except q, more than once. If the path ends at its start state q, it is an ε-cycle, and is inserted into $\widehat{C}_\varepsilon(q)$. All arcs e along a traversed path are put onto a stack (pseudo code, line 3: *push*(STACK, e)) so that at any time we can describe the path by the content of the stack (line 5: π=*path*(STACK)) :

```
T ⟶ T′ :
1     for ∀q ∈ Q
2         do STACK := {}
3            Ĉε(q) := {}
4            follow_epsilon_arcs(q)

follow_epsilon_arcs(p) :
1     for ∀e ∈ E(p)
2         do if i(e) = ε
3                then push(STACK, e)
4                     if n(e) = q
5                         then Ĉε(q) := Ĉε(q) ∪ {π | π=path(STACK)}
6                         else if ∀e′ ∈ path(STACK), n(e)≠n(e′)
7                                 then follow_epsilon_arcs(n(e))
8                     pop(STACK)
```

Although the $\widehat{C}_\varepsilon(q)$ do not contain all ε-cycles of a state q, the missing ε-cycles, that traverse a state q' more than once, do not escape our attention. They are in the $\widehat{C}_\varepsilon(q')$ of q' which is sufficient for our final purpose. The reason for building $\widehat{C}_\varepsilon(q)$ instead of $C_\varepsilon(q)$ is that $\widehat{C}_\varepsilon(q)$ is easier to construct, to represent (by an arc sequence), and to "rotate" (Sec 3.2).

3.2 Construction of T_1

Two steps are required to build T_1 from T' (Fig. 5) : First, at every state q_i with a non-empty set $\widehat{C}_\varepsilon(q_i)$, an arc mapping ε to ξ_i must be inserted. Second, all ε-cycles must be removed without physically removing their arcs.

$\mathbf{T'} \longrightarrow \mathbf{T'_1}$:
1 for $\forall q_i \in Q$
2 do if $\widehat{C}_\varepsilon(q_i) \neq \{\}$
3 then $Q := Q \cup \{q'_i\}$
4 $E := E \cup \{(q'_i, \varepsilon, \xi_i, q_i)\}$
5 for $\forall e \in E^R(q_i)$
6 do if $\widehat{C}_\varepsilon(q_i) \not\subseteq rotate_e^{LR}(\widehat{C}_\varepsilon(p(e)))$
7 then $n(e) := q'_i$

We insert for every state q_i with non-empty $\widehat{C}_\varepsilon(q_i)$, an auxiliary state q'_i and an auxiliary arc e'_i leading from q'_i to q_i (Fig. 6, dashed states and arcs). The arc e'_i is labeled with $\varepsilon{:}\xi_i$, i.e., it emits the auxiliary symbol ξ_i when it is traversed. For example, the auxiliary state $1'$ in created for state 1, and the auxiliary arc *200* labeled with $\varepsilon{:}\xi_1$ is inserted from state $1'$ to 1.

Then, some incoming arcs of every state q_i are redirected to the corresponding auxiliary state q'_i so that ξ_i is emitted before q_i is reached. An incoming arc e requires no redirection if the set $\widehat{C}_\varepsilon(q_i)$ of its destination state $n(e) = q_i$ is a "repetition", relative to e, of part of the $\widehat{C}_\varepsilon(p(e))$ of its source state $p(e)$. This is the case if every ε-cycle in $\widehat{C}_\varepsilon(q_i)$ can be obtained by "rotating" an ε-cycle in $\widehat{C}_\varepsilon(p(e))$, left to right, over e (pseudo code, line 6). In this case a redirection of e would not be wrong but it is redundant and can lead to a larger T_1 and T_2.

For example, the arc *106* requires no redirection from state 1 to $1'$ because every ε-cycle in $\widehat{C}_\varepsilon(1)$ can be obtained by rotating an ε-cycle in $\widehat{C}_\varepsilon(3)$ over the arc *106*; namely the ε-cycle ⌈*102, 105,* ***106***⌉ in $\widehat{C}_\varepsilon(1)$ by rotating ⌈***106***, *102, 105*⌉ in $\widehat{C}_\varepsilon(3)$ over the arc ***106***, and the ε-cycle ⌈*103,* ***106***⌉ in $\widehat{C}_\varepsilon(1)$ by rotating ⌈***106***, *103*⌉ in $\widehat{C}_\varepsilon(3)$ over the same arc ***106*** (Fig. 5, 6). In other terms, since ⌈*(106, 102, 105)*, 106*⌉=⌈*106, (102, 105, 106)**⌉ and ⌈*(106, 103)*, 106*⌉=⌈*106, (103, 106)**⌉, which in both cases means ⌈ξ_3*, 106*⌉=⌈*106,* ξ_1⌉, the insertion of ξ_1 after

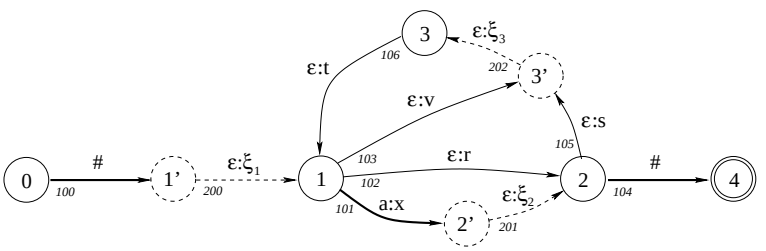


Fig. 6. Transducer T'_1 with redirected ε-arcs (Example 2)

the arc *106*, which would result from a redirection of this arc, is unnecessary; ξ_1 would not express anything that has not been described yet by ξ_3.

The arc *103* must be redirected from state 3 to 3′ because the ε-cycle ⌈*106, 102, 105*⌉ in $\widehat{C}_\varepsilon(3)$ cannot be obtained by rotating any of the ε-cycles in $\widehat{C}_\varepsilon(1)$ over the arc *103*. The arc *101* must be redirected from state 2 to 2′ because it is not an ε-arc which means that no ε-cycles can be rotated over it.

To prepare the removal of ε-cycles, the ε on the input side of every arc of every $\widehat{C}_\varepsilon(q_i)$ is temporarily overwritten by an auxiliary symbol ζ_j (Fig. 6, 7). This auxiliary symbol is different for every concerned arc, e.g., it is ζ_0 for the arc *102* and ζ_1 for the arc *105*. We call the result T_1''.

$\mathbf{T}_1' \longrightarrow \mathbf{T}_1''$:

```
1      j := 0
2      for ∀q ∈ Q
3          do for ∀e ∈̿ π ∈ Ĉε(q)
4                 do if i(e) = eps
5                        then i(e) := ζj
6                             j := j + 1
```

Every ε-cycle in T_1'' is then described by a sequence of ζ_j. For example, the ε-cycle ⌈*102, 105, 106*⌉ in $\widehat{C}_\varepsilon(1)$ is described by the sequence $\lceil \zeta_0, \zeta_1, \zeta_2 \rceil$ that consists of the new input symbols of this cycle (Fig. 5, 6, 7). Then, a constraint R_1 is formulated to disallow all ε-cycles in all sets $\widehat{C}_\varepsilon(q_i)$, by disallowing the corresponding ζ_j-sequences:

$$R_1 \;=\; \neg\Big(\; ?^* \; \Big(\bigcup_q i(\widehat{C}_\varepsilon(q)) \Big) \; ?^* \; \Big) \tag{7}$$

In Example 2, this constraint is (Fig. 7) :

$$R_1 \;=\; \neg\Big(\; ?^* \; \Big((\zeta_0\,\zeta_1\,\zeta_2) \cup (\zeta_3\,\zeta_2) \cup (\zeta_1\,\zeta_2\,\zeta_0) \cup (\zeta_2\,\zeta_0\,\zeta_1) \cup (\zeta_2\,\zeta_3) \Big) \; ?^* \; \Big) \tag{8}$$

example

When R_1 is composed on the input side of T_1'', all ε-cycles disappear; even those that are in $C_\varepsilon(q_i)$, but not in $\widehat{C}_\varepsilon(q_i)$, of a state q_i because they appear in $\widehat{C}_\varepsilon(q_k)$ of at least one other state q_k:

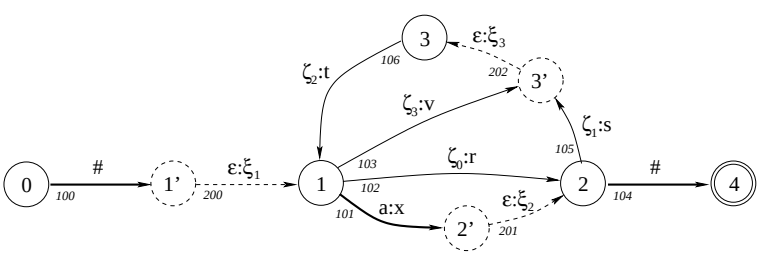


Fig. 7. Transducer T_1'' with redirected and overwritten ε-arcs (Example 2)

$$T_1''' \quad = \quad R_1 \diamondsuit T_1'' \tag{9}$$

However, instances of the ζ_j-arcs remain in T_1''' if they are also part of another path that is not an ε-cycle. Finally, every remaining ζ_j, which stands for ε, and every boundary symbol, #, which has to be removed, is replaced with ε, and T_1''' is minimized (Fig. 9a). We call the result T_1. Note that an initially introduced auxiliary symbol ξ_i does no appear in T_1 if none of the incoming arcs of the state q_i have been redirected.

3.3 Construction of T_2

T_2 is built from T' (as was the case with T_1) (Fig. 5). T_2 must map any auxiliary symbol ξ_i to the corresponding set of ε-cycles $C_\varepsilon(q_i)$ rather than $\widehat{C}_\varepsilon(q_i)$. For every state q_i with non-empty $\widehat{C}_\varepsilon(q_i)$, two auxiliary arcs, both labeled with the auxiliary symbol ξ_i, are created (Fig. 8); one arc leading from the initial state i to q_i, the other from q_i to the only final state f (pseudo code, lines 3 and 4). The resulting FST will be referred to as T_2'.

$\mathbf{T'} \longrightarrow \mathbf{T_2'}$ **:**

```
1 for ∀q_i ∈ Q
2     do if Ĉ_ε(q_i) ≠ {}
3             then E := E ∪ {⟨i, ξ_i, ξ_i, q_i⟩}
4                  E := E ∪ {⟨q_i, ξ_i, ξ_i, f⟩}
```

All paths in T_2' that contain only full (and no partial) ε-cycles of a state q_i must be kept and all others removed. For example, the set of paths $\lceil$*301, (102, 105, 106)*, 304*$\rceil$ containing all ε-cycles of $C_\varepsilon(1)$ must be kept and $\lceil$*301, (102, 105, 106)*, 102, 305*$\rceil$ must be removed (Fig. 8). The paths to be kept, consist of twice the same auxiliary symbol, $\xi_i\,\xi_i$, on the input side. To allow only them, T_2' is composed with a constraint:

$$T_2'' \quad = \quad \left(\bigcup_i \{\xi_i\ \xi_i\} \right) \diamondsuit T_2' \tag{10}$$

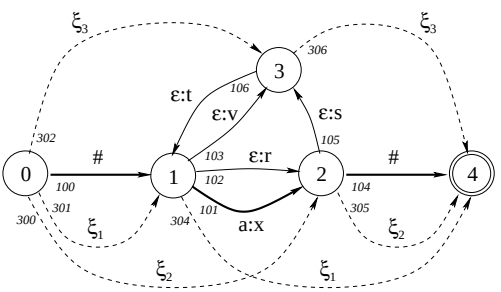


Fig. 8. Transducer T_2' (Example 2)

This removes all undesired paths. In Example 2, the composition is (Fig. 8) :

$$T_2'' \;=\; \Big((\xi_1\ \xi_1) \;\cup\; (\xi_2\ \xi_2) \;\cup\; (\xi_3\ \xi_3) \Big) \diamondsuit T_2' \qquad (11)$$

example

The resulting T_2'' maps any sequence of two identical auxiliary symbols $\xi_i\xi_i$ to itself, and inserts the corresponding set of ε-cycles $C_\varepsilon(q_i)$ in between. The second occurrence of every ξ_i is actually unwanted. The following composition removes this second occurrence on the input and output side, and the first occurrence of ξ_i on the output side only:

$$T_2''' \;=\; (\ ?\ \ \hat{\varepsilon}{:}?\ )\ \diamondsuit\ T_2''\ \diamondsuit\ (\ ?{:}\varepsilon\ \ ?^*\ \ ?{:}\hat{\varepsilon}\ ) \qquad (12)$$

The resulting T_2''' maps any single auxiliary symbol ξ_i to the corresponding set $C_\varepsilon(q_i)$. The $\hat{\varepsilon}$ denotes the (ordinary) empty string, like ε. It is, however, preserved in minimization and determinization which prevents T_2 from becoming larger. If the size is of no concern, ε can be used instead.

T_2 must accept any sequence of output symbols of T_1, i.e., any sequence in $\Delta^*_{T_1}$. It must map every auxiliary symbol ξ_i to the corresponding set of ε-cycles $C_\varepsilon(q_i)$, and every other symbol to itself. T_2 is built by:

$$T_2 \;=\; \left(\Delta_{T_1} \diamondsuit \left(T_2''' \;\cup\; \neg \bigcup_i \xi_i \right) \right)^* \qquad (13)$$

This operation has the side effect that all initially introduced auxiliary symbols ξ_i that later disappeared from T_1, are now also removed from T_2. Finally, T_2 is minimized (Fig. 9b).

3.4 Proof

For the following reason, the algorithm always leads to the described result.

In T_1: If an ε-cycle in $C_\varepsilon(q_i)$ contains a state q_k more that once, which means that this cycle has an "inner" ε-cycle on q_k, than $C_\varepsilon(q_i)$ also contains an ε-cycle where no q_k is encountered more that once, which can be obtained by not traversing the inner cycle on q_k. This also holds if there are several inner cycles. In general:

$$C_\varepsilon(q_i) \neq \{\} \quad \Rightarrow \quad \widehat{C}_\varepsilon(q_i) \neq \{\} \qquad (14)$$

This means, every q_i with $C_\varepsilon(q_i) \neq \{\}$ will be assigned an auxiliary symbol ξ_i, although this action is triggered by $\widehat{C}_\varepsilon(q_i) \neq \{\}$.

All inner ε-cycles, that are not in $\widehat{C}_\varepsilon(q_i)$, are in $\widehat{C}_\varepsilon(q_k)$, of some other state q_k. Consequently, they will be removed as well, i.e., all ε-cycles of T_1 will be removed.

Example 2 (Fig. 5) : Since state 2 has a non-empty $C_\varepsilon(2) = \{\lceil$*105, (106, 103)*, 106, 102*$\rceil\}$, it also has a non-empty $\widehat{C}_\varepsilon(2) = \{\lceil$*105, 106, 102*$\rceil\}$ and will therefore be assigned ξ_2. The inner cycle $\lceil$*(106, 103)**$\rceil$ in $C_\varepsilon(2)$ will be removed from T_1, despite not being in $\widehat{C}_\varepsilon(2)$, because it is in $\widehat{C}_\varepsilon(1)$ and $\widehat{C}_\varepsilon(3)$.

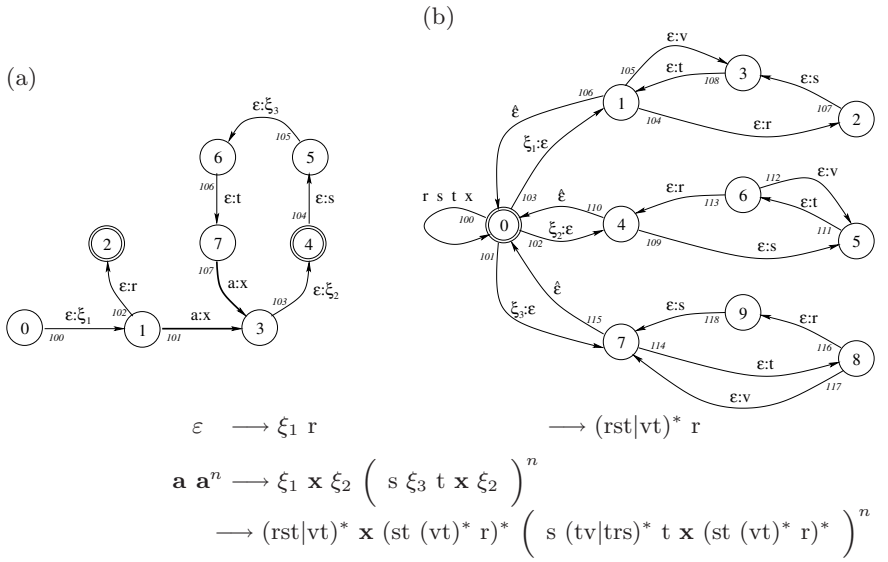


Fig. 9. Factorization of T with ε-cycles into (a) T_1 that emits auxiliary symbols, and (b) T_2 that maps auxiliary symbols to ε-cycles (Example 2)

In T_2: The $C_\varepsilon(q_i)$ of every state q_i that has been assigned an auxiliary symbol ξ_i are preserved whereas every other path is removed. This means, the ξ_i are mapped to $C_\varepsilon(q_i)$ rather than to $\widehat{C}_\varepsilon(q_i)$ that originally caused their introduction.

The initial limitation to $\widehat{C}_\varepsilon(q_i)$ is not reflected in the final result, T_1 and T_2.

4 Final Remarks

Although T_1 (Fig. 9a) cannot be converted into a single real-time FST, it can be split into the union of two FSTs, one containing the transitions {100, 102}, the other the transitions {100, 101, 103, 104, 105, 106, 107}. The second of these FSTs can be made real-time.

Jointly in a cascade, T_1 and T_2 describe the same relation and perform the same mapping as the original FST T (Fig. 9). When T_1 and T_2 are composed with each other, T is obtained. The size increase of T_2, compared to T, is not necessarily a concern. T_2 could be an intermediate result that is further processed.

As previously show for Example 1, instead of factorizing T into T_1 and T_2, one can represent T by a single FST, $\widehat{T}$, that is similar to T_1 but with output labels directly describing sets of ε-cycles $C_\varepsilon(q)$. Every $C_\varepsilon(q)$ in T would be reduced to a single arc in $\widehat{T}$ (Fig. 10). Both representations are equivalent and require, as mentioned, an algorithm (possibly very similar) to construct them.

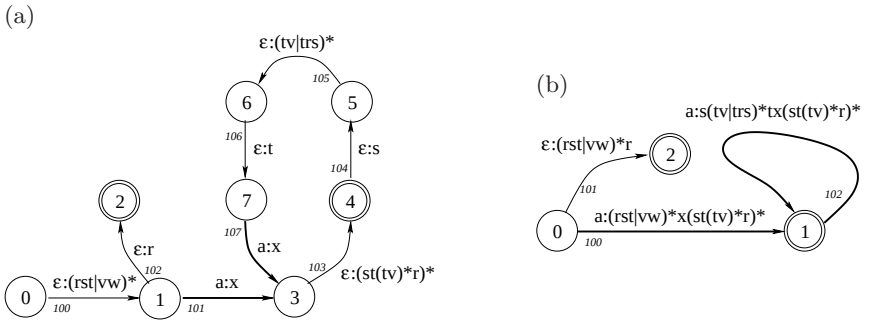


Fig. 10. Alternative representation of ε-cycles by complex labels (a) with ε-arcs or (b) more compact with less ε-arcs (Example 2)

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, USA. 190

[2] G. Birkhoff and T. C. Bartee. 1970. *Modern Applied Algebra.* McGraw-Hill, New York, USA. 191

[3] A. Kempe. 2000. Reduction of intermediate alphabets in finite-state transducer cascades. In *Proceedings of the 7th Conference on Automatic Natural Language Processing (TALN)*, pages 207–215, Lausanne, Switzerland. ATALA. 193

[4] A. Kempe. 2001. Factorization of ambiguous finite-state transducers. In S. Yu, A. Paun, editors, *Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA 2000)*, The University of Western Ontario, London, Ontario, Canada, July 24-25, 2000. Volume 2088 of *Lecture Notes in Computer Science*, pages 170–181, Springer-Verlag. 190

[5] M. Mohri. 2001. Generic ε-removal algorithm for weighted automata. In S. Yu, A. Paun, editors, *Proceedings of the 5th International Conference on Implementation and Application of Automata (CIAA 2000)*, The University of Western Ontario, London, Ontario, Canada, July 24-25, 2000. Volume 2088 of *Lecture Notes in Computer Science*, pages 230–242, Springer-Verlag. 190

[6] G. van Noord. 1998. Treatment of ε-moves in subset construction. In *Proceedings of the International Workshop on Finite-State Methods in Natural Language Processing (FSMNLP)*, pages 1–12, Ankara, Turkey, June 29 - July 1. Bilkent University. 190

[7] J. Sakarovitch. 1998. A construction on finite automata that has remained hidden. *Theoretical Computer Science*, 204:205–231. 190, 193

[8] M. P. Schützenberger. 1976. Sur les relations rationnelles entre monoïdes libres. *Theoretical Computer Science*, 3:243–259. 190

On the Size of Deterministic Finite Automata*

Bořivoj Melichar[1] and Jan Skryja[2]

[1] Department of Computer Science and Engineering, Czech Technical University
Karlovo nám. 13, CZ-121 35, Prague 2, Czech Republic
melichar@fel.cvut.cz
[2] Aquasoft
Těšnov 5, 110 01 Prague 1, Czech Republic
j.skryja@aquasoft.cz

Abstract. The number of states of a deterministic finite automaton, which is equivalent to a nondeterministic finite automaton is bounded by 2^n, where n is the number of states of the nondeterministic finite automaton. This bound is very pesimistic. Some subclasses of finite automata are shown, for which the complexity of determinization is far lower. On the base of pioneering idea of homogenous finite automata, two classes of finite automata are defined and the complexity of their determinization are shown. These classes are called generalized homogenous finite automata and semihomogenous finite automata.

1 Introduction

It is known for a long time, that the number of states of a deterministic finite automaton which is equivalent to a nondeterministic finite automaton is bounded by 2^n, where n is the number of states of the nondeterministic finite automaton. But this exponential complexity is reached in a very few cases. The complexity is mostly far lower than exponential. This fact is a challenge for researchers. They are trying to find the upper bound of this complexity for some subclasses of finite automata. This effort results in finding many subclasses of finite automata having lower complexity of the determinization. Let us mention factor automata [RS97b], subsequence automata and searching automata [H00] as examples. But these classes of automata are very small and oriented to a particular applications. Therefore the research is oriented to greater classes of finite automata. Pioneering idea was the definition of homogenous finite automata [CZP99]. For this class of automata the more realistic bound of determinization was found. This paper uses the notion of homogenous finite automata and reports two possible extensions of this notion. The resulting classes of automata are generalized homogenous finite automata and semihomogenous finite automata. The complexity of determinization is greater for these classes than for the homogenous finite automata but lower than the general bound. Some ideas and examples in this paper are taken from diploma thesis [S00].

* This research was partially suported by grants 201/98/1155 and 201/01/1433 of the Grant Agency of Czech Republic.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 202–213, 2002.
© Springer-Verlag Berlin Heidelberg 2002

After the introduction of basic notions and notations in Section 2 and the general approach to determinization in Section 3, the class of homogenous finite automata is defined and the complexity of the determinization is shown in Section 4. Sections 5 and 6 introduce the notions of generalized homogenous finite automata and semihomogenous finite automata, respectively.

2 Basic Notions and Notations

The notions used in this paper are in common use in the clasical theory of finite automata and can be found in almost all textbooks.

An *alphabet* is a finite set of symbols. The *set of strings* over an alphabet T is denoted by T^* including empty string denoted by ε.

A *nondeterministic finite automaton* (*NFA*) M is a quintuple $M = (Q, T, \delta, q_0, F)$, where Q is the finite set of states, T is the input alphabet, $\delta : Q \times T \to \mathcal{P}(Q)$ is the mapping called the state transition function, $q_0 \in Q$ is the initial state, and $F \subset Q$ is the set of final states.

A *deterministic finite automaton* (*DFA*) is a special case of *NFA*, where the mapping $\delta : Q \times T \to Q$ is its state transition function.

We will use abbreviation *FA* for both *NFA* and *DFA*. A pair $(q, w) \in Q \times T^*$ is a configuration of *FA* M. A relation $\vdash \in (Q \times T^*) \times (q \times T^*)$ is called a *transition* of *FA* M. The *transitive and reflexive closure* of the relation $\vdash$ is denoted as $\vdash^*$. Let us have *FA* $M = (Q, T, \delta, q_0, F)$. A state $q \in Q$ we call an *accessible state* if such string $w \in T^*$ exists that the sequence of transitions exists leading from the initial state to the state q. Other states are inaccessible. The state $q \in Q$ we call *useless* if no sequence of transitions from q leads to some final state. The *size of FA* is the number of its accessible states, which are not useless.
The language accepted by *FA* $M, L(M)$ is the set $L(M) = \{w : w \in T^*, (q_0, w) \vdash^* (q, \varepsilon), q \in F\}$. The *FA* is complete, if $\delta(q, a)$ is defined for all $a \in T$ and for all $q \in Q$.

The *cardinality* of a set X is denoted as $|X|$. $\mathcal{P}(X)$ is the *powerset* of set X. A *cover of set* Q is a collection of sets $C(Q) = \{Q_1, Q_2, \ldots, Q_k\}$ such that

$$Q = \bigcup_{i=1}^{k} Q_i, \ Q_i \subset Q \qquad \text{for} \quad i = 1, 2, \ldots, k.$$

A *decomposition of set* Q is a cover in which all sets are pairwise disjoint.

$$D(Q) = \{Q_i\}_i^k = \bigcup_{i=1}^{k} Q_i.$$

Sets in the decomposition $D(Q)$ are called classes.

3 General Approach

The general relation between the nondeterministic and deterministic finite automata is given the following Theorem.

Theorem 3.1
For each *NFA* M there exists a complete *DFA* M' such that $L(M) = L(M')$. Moreover, if the automaton M has n states, then automaton M' has at most 2^n states. □

This Theorem is contained in almost all books on finite automata theory, for example in [RS 97]. It follows from so called subset construction performed by the following Algorithm.

Algorithm 3.2
Construction of *DFA*.
Input: *NFA* $M = (Q, T, \delta, q_0, F)$
Output: *DFA* $M' = (Q', T, \delta', q'_0, F')$ such that $L(M) = L(M')$
Method:

1. $Q' = \mathcal{P}(Q)$, the set of all subsets of Q.
2. $\delta'(q', a) = \bigcup \delta(p, a)$ for all $p \in q'$ and for $a \in T$.
3. $q'_0 = \{q_0\}$.
4. $F' = \{q' : q' \in Q', q' \cap F \neq \emptyset\}$. □

This Algorithm produces *DFA* having 2^n states for *NFA* having n states. The reason is that the number of all subsets of the set Q is just 2^n.

Example 3.3
Example of maximum size of *DFA*. Let us have *NFA*
$M = (\{1, 2, 3\}, \{a, b\}, \delta, 1, \{3\})$, where δ is specified as follows:

δ	a	b
1	2	$\emptyset$
2	3	1, 2
3	1	1, 3

This automaton is nondeterministic and has 3 states. The equivalent *DFA* has $2^3 = 8$ states. $M' = (Q', \{a, b\}, \delta', \{1\}, F')$, where δ' is specified as follows:

δ'	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{1\}$	$\{2\}$	$\emptyset$
$\{2\}$	$\{3\}$	$\{1, 2\}$
$\{3\}$	$\{1\}$	$\{1, 3\}$
$\{1, 2\}$	$\{2, 3\}$	$\{1, 2\}$
$\{1, 3\}$	$\{1, 2\}$	$\{1, 3\}$
$\{2, 3\}$	$\{1, 3\}$	$\{1, 2, 3\}$
$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{1, 2, 3\}$

where F' contains states $\{1,3\}, \{2,3\}, \{1,2,3\}$ and $\{3\}$.

This example shows that the determinization of *NFA* can result to above-mentioned exponencial size of equivalent *DFA*. However in most cases not all subsets are necessary. Thus, it suffices to construct only those subsets that are accessible from the inital state. □

Example 3.4
Let us have *NFA* $M = (\{1,2,3,4\}, \{a,b\}, \delta, 1, \{4\})$, where δ is specified as follows:

δ	a	b
1	1, 2	1
2	$\emptyset$	3
3	$\emptyset$	4
4	$\emptyset$	$\emptyset$

This automaton is nondeterministic and has 4 states. The equivalent *DFA* has 4 accessible states. $M' = (Q', \{a,b\}, \delta', \{1\}, F')$, where δ' is specified as follows:

δ'	a	b
$\{1\}$	$\{1,2\}$	$\{1\}$
$\{1,2\}$	$\{1,2\}$	$\{1,3\}$
$\{1,3\}$	$\{1,2\}$	$\{1,4\}$
$\{1,4\}$	$\{1,2\}$	$\{1\}$

This example shows, that the exponential size of the determinization is pesimistic and some improvements of the estimation of this size will be usefull.

□

4 Homogenous Finite Automata

Homogenous finite automaton is such finite automaton, where for all states the following condition holds:

Transitions leading to a particular state are all for the same symbol of the input alphabet.

The determinization of homogenous *NFA* leads to *DFA* having lower size than in the general case.

Definition 4.1
Let $M = (Q, T, \delta, q_0, F)$ be a finite automaton. Let us define for arbitrary $a \in T$ the set $Q(a) \subset Q$ as follows:

$$Q(a) = \{q : q \in \delta(p,a), a \in T, p, q \in Q\}.$$ □

Definition 4.2
Let $M = (Q, T, \delta, q_0, F)$ be a finite automaton and $Q(a)$ be sets of states for all symbols $a \in T$. If for all pairs of symbols $a, b \in T$, $a \neq b$, it holds $Q(a) \cap Q(b) = \emptyset$, then the automaton M is called homogenous. The collection of sets $\{Q(a) : a \in T\}$ is for the homogenous finite automaton the decomposition on classes having one of these two forms:

1. $Q = \bigcup_{a \in T} Q(a) \cup \{q_0\}$ in case that $q_0 \notin \delta(q, a)$ for all $q \in Q$ and all $a \in T$,
2. $Q = \bigcup_{a \in T} Q(a)$ in case that $q_0 \in \delta(q, a)$ for some $q \in Q, a \in T$. In this case $q_0 \in Q(a)$. □

The following Theorem is taken from [CZP99].

Theorem 4.3
Let $M = (Q, T, \delta, q_0, F)$ be a homogenous *NFA*. Then the size of the equivalent *DFA* $M' = (Q', T, \delta', q_0', F')$ is as follows:

$$|Q'| \leq \sum_{a \in T} (2^{|Q(a)|}) - |T| + 1.$$ □

Proof
DFA $M' = (Q', T, \delta', q_0', F')$ can have as accesible states only the subsets of $Q(a)$ for all $a \in T$ and q_0'. The number of such states for symbol a is at most $2^{|Q(a)|}$. It follows that for all input symbols the number of such subsets is $\sum_{a \in T} (2^{|Q(a)|})$. Empty set is one of such subsets for each input symbol. This empty set represents the useless state. Therefore we substract $|T|$, which is the number of such subsets in the abovementioned sum. Moreover, we add 1 because there need not exist a transition to the initial state q_0 and therefore state q_0 need not be the element of any $Q(a)$ but it appears in *DFA*. Then it holds

$$|Q'| \leq \sum_{a \in T} (2^{(Qa)}) - |T| + 1.$$ □

This bound is generally much lower than the general bound.

Algorithm 3.2 constructs *DFA* starting with the construction of the powerset of Q. Therefore the intermediate result for homogenous *NFA* has the maximal complexity. Many of the created states are inaccessible or useless. In order to overcome this problem we present algorithm for the construction of *DFA* that creates only accessible states.

Algorithm 4.4
Transformation of an *NFA* to the equivalent *DFA* without inaccessible states.
Input: A finite automaton $M = (Q, T, \delta, q_0, F)$.
Output: A deterministic finite automaton $M' = (Q', T, \delta', q_0, F')$ such that $L(M) = L(M')$.
Method:

1. The set $Q' = \{\{q_0\}\}$ will be defined, the state $\{q_0\}$ will be treated as unmarked.
2. If each state Q' is marked, then continue with step (4).
3. An unmarked state q' will be chosen from Q' and the following operations will be executed:

(a) We will determine $\delta'(q', a) = \cup\delta(p, a)$ for $p \in q'$ and for all $a \in T$,
(b) $Q' = Q' \cup \delta'(q', a)$ for all $a \in T$,
(c) the state $q' \in Q'$ will be marked,
(d) continue with step 2.

4. $q'_0 = \{q_0\}$.
5. $F' = \{q' : q' \in Q', q' \cap F \neq \emptyset\}$. □

Example 4.5
Let us have homogenous *NFA* $M = (\{p, q, r, s, t, u\}, \{a, b\}, \delta, p, \{r, u\})$,

where the mapping δ is defined by the transition table:

	a	b
p	q	s
q	q, r	s, t
r		u
s	q	u
t	r	s
u		t

In this case it holds:

$$Q(a) = \{q, r\},\ Q(b) = \{s, t, u\}.$$

These sets have empty intersection and thus *DFA* $M = (Q', T, \delta', q_0, F')$ has the number of states less or equal to

$$2^{|Q(a)|} + 2^{|Q(b)|} - |T| + 1 = 2^2 + 2^3 - 2 + 1 = 4 + 8 - 2 + 1 = 11.$$

The equivalent *DFA* is

$$M' = (\{p, q, qr, s, t, st, stu, u, su, r, tu\}, \{a, b\}, \delta, p, \{r, qr, u, su, tu, stu\}),$$

where the mapping δ is defined by the transition table:

	a	b
p	q	s
q	qr	st
s	q	u
qr	qr	stu
st	qr	su
u		t
stu	qr	stu
su	q	tu
t	r	s
tu	r	st
r		u

It is seen that the resulting *DFA* have just 11 states. □

5 Generalized Homogenous Finite Automata

The notion of the homogenous automata is possible to generalize. One of possible generalizations leads to the situation, where the transitions leading to the particular state are all for the some subset of the input alphabet. If sets $Q(a)$ for $a \in T$ are not pairwise disjoint, we can try to find the set of symbols for which transitions are going only to the particular subset of states. It means that we are trying to find some decomposition of the set of states, which is not so fine like for homogenous automata. Every set in this decomposition corresponds to some subset of states and these subsets are pairwise disjoint.

Definition 5.1
Let us have a finite automaton $M = (Q, T, \delta, q_0, F)$. We define for $T' \subset T$ the set $Q(T') \subset Q$ as follows:

$$Q(T') = \bigcup_{a \in T'} Q(a).$$

Definition 5.2
Let us have a finite automaton $M = (Q, T, \delta, q_0, F)$ and a cover $\{T_i\}_1^k,\ k \geq 2$, of an input alphabet T is such that in the corresponding collection of sets $\{Q(T_i)\}_1^k$ are all disjoint. Then the finite automaton M is a generalized homogenous finite automaton.

Theorem 5.3
Collection of sets $\{Q(T_i)\}_1^k$ for a generalized homogenous finite automaton is a decomposition of set of states on classes possibly extended with the set $\{q_0\}$.

Proof
It is enough to prove, that the collection of sets $Q(T_i)$ is a cover of the set of states, i.e. $Q(T_1) \cup Q(T_2) \cup \ldots \cup Q(T_k) \cup \{q_0\} = Q$.
The proof we make by contradiction. Let state $q \neq q_0$ is an element of no set $Q(T_i)$. But the state q is neither inaccessible nor useless. Thus some transition is going to it for some input symbol $a \in T$. Then there exists some i such that $a \in T_i$ and therefore $q \in Q(T_i)$, which is the contradiction. □

Theorem 5.4
Let $M = (Q, T, \delta, q_0, F)$ is a finite automaton and $T', T'' \subset T$ are subsets of input alphabet such that $Q(T') \cap Q(T'') = \emptyset$. Then $T' \cap T'' = \emptyset$.

Proof
We make the proof by contradiction. Let us suppose that symbol $a \in T' \cap T''$. Then $Q(a) \subset Q(T')$ and $Q(a) \subset Q(T'')$. Thus $Q(a) \subset Q(T') \cap Q(T'')$. Then $Q(a) \neq \emptyset$ and therefore $Q(T') \cap Q(T'') \neq \emptyset$, which is the contradiction. □

There exists a decomposition of the set of states of generalized homogenous finite automaton having the form

$$Q = Q(T_1) \cup Q(T_2) \cup \ldots \cup Q(T_k) \cup \{q_0\},$$

when $q_0 \in \delta(q, a)$ for no $q \in Q,\ a \in T$ or

$$Q = Q(T_1) \cup Q(T_2) \cup \ldots \cup Q(T_k),$$

otherwise. This decomposition further induces decomposition of the set of input symbols:

$$T = T_1 \cup T_2 \cup \ldots \cup T_k.$$

Theorem 5.5
Let $M = (Q, T, \delta, q_0, F)$ be a generalized homogenous *NFA*. Then the number of states of the equivalent *DFA* $M' = (Q', T, \delta', q_0, F')$ is

$$|Q'| \leq \sum_{i=1}^{k} (2^{|Q(T_i)|}) - k + 1,$$

where k is the number of classes of the set of states and therefore the set of input symbols, too. □

Proof
DFA $M' = (Q', T, \delta', q_0, F')$ can have as accessible states only subsets of sets $Q(T_i)$ for all $T_i \subset T$, $i = 1, 2, \ldots, k$. The number of such states for set T_i is at most $2^{|Q(T_i)|}$. It follows that for all sets from the abovementioned decomposition the number of such subset

$$\sum_{i=1}^{k} (2^{|Q(T_i)|}).$$

One of possible subsets is the empty set for all sets of the decomposition. Therefore we substract k, which is the number of these subsets in the sum. Further we add 1 because there need not exist a transition to the initial state q_0 and therefore the state q_0 need not be the element of some set $Q(T_i)$ but in *DFA* it appears. In case, when the number of classes of the decomposition is just one, then the general estimation of number of states takes place

$$|Q'| \leq 2^{|Q|}.$$ □

Example 5.6
Let us have generalized homogenous *NFA* $M = (\{p, q, r, s\}, \{a, b, c, d\}, \delta, p, \{q, s\})$, where the mapping δ is defined by the following transition table:

	a	b	c	d
p	p, q		r, s	
q	p			s
r		q	r	
s				

It holds

$$Q(\{a, b\}) = \{p, q\}, Q(\{c, d\}) = \{r, s\}.$$

These sets have empty intersection and thus *DFA* will have the number of states less or equal to

$$2^{|Q(\{a,b\})|} + 2^{|Q(\{c,d\})|} - 2 + 1 = 2^2 + 2^2 - 2 + 1 = 4 + 4 - 2 + 1 = 7.$$

The resulting automaton will have six states only, because the initial state

$p \in \delta(q,a)$. The equivalent *DFA* is $M' = (\{p,q,pq,r,s,rs\},\{a,b,c,d\},\ \delta,p,$ $\{q,pq,s,rs\})$, where the mapping δ is given by the following transition table:

	a	b	c	d
p	pq		rs	
q	p			s
pq	pq		rs	s
r		q	r	
s				
rs		q	r	

□

Algorithm 5.7
Construction of the decomposition of the set of states of *FA*.
Input: *FA* $M = (Q,T,\delta,q_0,F)$.
Output: Decomposition $D(Q)$ of set of states Q on classes.
Method:

1. Create sets $\quad Q(a) = \bigcup_{q \in Q} \delta(q,a) \qquad$ for all $a \in T$.
2. If all sets $Q(a)$ are pairwise disjoint, then the decomposition is finished and Algorithm finishes. Automaton M is homogenous.
   $D(Q) = \{Q(a) : a \in T\}$.
3. In other case perform these operations:
   (a) $S(Q) = \{Q(a) : a \in T\}$.
   (b) $D(Q) = \emptyset$.
   (c) If $S(Q) = \emptyset$, the Algorithm finishes.
   (d) Select some set $Q(a)$ from $S(Q)$ and in addition also all sets $Q(b_i),\ i = 1,2,\ldots,k$, having a nonempty intersection with $Q(a)$ and $Q(b_1),\ldots,$ $Q(b_{i-1})$. $D(Q) = D(Q) \cup \{Q(a) \cup Q(b_1) \cup Q(b_2) \cup \ldots \cup Q(b_k)\}$.
   (e) Sets $Q(a), Q(b_i),\ i = 1,2,\ldots k$ remove from set $S(Q)$.
   (f) Repeat from step 3c). □

Example 5.8
Let as show the construction of the decomposition using *NFA* from Example 5.6.

1. $Q(a) = \{p,q\}$,
   $Q(b) = \{q\}$,
   $Q(c) = \{r,s\}$,
   $Q(d) = \{s\}$.
2. Automaton is not homogenous.
3. *a)* $S(Q) = \{Q(a),Q(b),Q(c),Q(d)\}$.
   b) $D(Q) = \emptyset$.
   c) $S(Q) \neq \emptyset$.
   d) $D(Q) = \emptyset \cup \{Q(a) \cup Q(b)\} = \{\{p,q\}\}$.
   e) $S(Q) = \{Q(c),Q(d)\}$.
3. *c)* $S(Q) \neq \emptyset$,
   d) $D(Q) = D(Q) \cup \{Q(c) \cup Q(d)\} = \{\{p,q\},\{r,s\}\}$.
   e) $S(Q) = \emptyset$. □

6 Semihomogenous Finite Automata

The notion of homogenous automata is possible to generalize using another approach than in the previous section. This generalization is based on the idea that instead of construction of decomposition of Q on classes we create a cover of the set of states according to a collection of subsets of set of states.

This cover has again two possibilities like in the decomposition in the Definition 4.2. It concerns of the initial state. Let us take more general case, when there is no transition going to the initial state. In this case

$$Q = \bigcup_{a \in T} Q(a) \cup \{q_0\}.$$

It can hold that $Q(a) \cap Q(b) \neq \emptyset$ for $a \neq b$. Thus we obtain the initial cover which need not be the optimal one.

Let us suppose that it holds for sets $Q(a)$ and $Q(b)$ that $Q(b) \subset Q(a)$. In such case subsets of set $Q(b)$ are also subsets of set $Q(a)$. If means that we need not take set $Q(b)$ into account. It follows that an optimal cover contains only such set from the initial cover, which are not subsets of other sets.

Definition 6.1
Let us have *FA* $M = (Q, T, \delta, q_0, F)$ and a cover of set of states $C(Q) = \{Q(a) : a \in T\}$. This cover we call optimal one if for no pair $a, b \in T$ holds $Q(b) \subset Q(a)$.

Algorithm 6.2
Construction of the optimal cover of the set of states of *FA*
Input: *FA* $M = (Q, T, \delta, q_0, F)$.
Output: Optimal cover $C(Q)$ of the set of states.
Method:

1. Create sets $Q(a) = \bigcup_{q \in Q} \delta(q, a)$ for all $a \in T$.
2. $C(Q) = \{Q(a) : a \in T\}$.
3. For all sets $X \in C(Q)$ perform this operation:
   If $X \subset Y$ for some set $Y \in C(Q)$, then leave out set X from $C(Q)$.

□

Definition 6.3
FA $M = (Q, T, \delta, q_0, F)$ we call semihomogenous if the optimal cover $C(Q)$ constains at least two sets.

Theorem 6.4
Let $M = (Q, T, \delta, q_0, F)$ be a semihomogenous *NFA* and the optimal cover $C(Q) = \{Q(a_1), Q(a_2), \ldots, Q(a_n)\}$. Then the number of states of the equivalent *DFA* $M' = (Q', T, \delta', q_0, F')$ is given by

$$|Q'| \leq \sum_{i=1}^{n}(2^{|Q(a_i)|}) - n - \sum_{i=1}^{n}\sum_{j=i+1}^{n}(2^{|Q(a_i)\cap Q(a_j)|}) + \frac{n^2-n}{2} + 1$$

Proof
DFA $M' = (Q', T, \delta', q_0, F')$ can have as accessible states only subsets of sets $Q(T_i)$ for all $T_i \subset T,\ i = 1, 2, \ldots, k$. The number of such states for set T_i is at most $2^{|Q(T_i)|}$. It follows that for all sets from the abovementioned decomposition the number of such subset $\sum_{i=1}^{k}(2^{|Q(T_i)|})$.
One of possible subsets is the empty set for all sets of the decomposition. Therefore we substract n, which is the number of these subsets in the sum.
In case when two subsets $Q(a)$ and $Q(b)$ have nonempty intersection $X = Q(a) \cap Q(b)$, then subsets of X are taken into account two times. Therefore we can substract the number of subsets of this intersection. The number of these subsets is

$$2^{|X|} = 2^{|Q(a)\cap Q(b)|}.$$

Thus we subsctract the number of subsets created by intersection operation which is

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n}(2^{|Q(a_i)\cap Q(a_j)|}).$$

Futher, we must add $\frac{n^2-n}{2}$, which is the number of empty sets in the sum above. The number of all subsets is

$$\sum_{i=1}^{n}(2^{|Q(a_i)|}) - n - \sum_{i=1}^{n}\sum_{j=i+1}^{n}(2^{|Q(a_i)\cap Q(a_j)|}) + \frac{n^2-n}{2} + 1.$$

We add one in the case if the initial state is element of no set $Q(a_i)$. □

Example 6.5
Let us have *NFA* $M = (\{p, q, r, s\}, \{a, b, c, d\}, \delta, p, \{s\})$, where the mapping δ is given by the transition table:

	a	b	c	d
p	q	p, q		
q	r	r	r	
r	p		r, s	s
s			r	

The initial cover is:

$$Q(a) = \{p, q, r\}, Q(b) = \{p, q, r\}, Q(c) = \{r, s\}, Q(d) = \{s\}.$$

This cover is not optimal and after removing $Q(a)$ (it is the subset of $Q(b)$) and $Q(d)$ (it is the subset of $Q(c)$) we get the optimal cover:

$$C(Q) = (\{p, q, r\}, \{r, s\}).$$

The number of states of *DFA* will be:

$$|Q'| \leq 2^{|Q(b)|} + 2^{|Q(c)|} - 2 - 2^{|Q(b)\cap Q(c)|} + 1 + 1$$
$$= 2^3 + 2^2 - 2 - 2^1 + 1 + 1 = 8 + 4 - 2 - 2 + 1 + 1 = 10$$

The resulting automaton has nine states because $p \in Q(b)$. *DFA* is

$$M' = (\{p, q, pq, pr, qr, pqr, r, s, rs\}, \{a, b, c, d\}, \delta, p, \{s, rs\}),$$

where the mapping δ is given by the transition table:

	a	b	c	d
p	q	pq		
q	r	r	r	
pq	qr	pqr	r	
qr	pr	r	rs	s
pqr	pqr	pqr	rs	s
pr	pq	pq	rs	s
r	p		rs	s
s			r	
rs	p		rs	s

□

7 Conclusion

We have presented two classes of *FA* and we have shown how to compute the size of *DFA*, which is the result of determinization of the equivalent *NFA*. Both classes of these automata, generalized homogenous *FA* and semihomogenous *FA*, are defined as a generalization of the class of homogenous *FA*. The size of the determinazition of these two classes of *FA* is less pesimistic like in the general case. Nevertheless, this size is still exponential, but in both cases it is the sum of exponetial functions. This sum is always lower than in the general case.

This result can be used as one of sources for computation of upper bound of size of some classes of automata, because in many aplications (generalized) homogenous and/or semihomogenous *NFA* are usefull.

References

[CZP99] Champarnaud, J. M., Ziadi, D., Ponty, J. L.: *Determinization of Glushhov Automata In: Champarnaud, J. M., Maurel, D., Ziadi, D. (eds.): Automata Implementation.* LNCS 1660 Heidelberg, Springer–Verlag 1999.

[H00] Holub, J.: *Simulation of Nondeterministic Finite Automata in Pattern Matching.* Ph. D. Thesis, Czech Technical University, Prague, Czech Republic, 2000.

[RS97a] Rozenberg, G., Salomaa, A., (eds.): *Handbook of Formal Languages.* Springer, Berlin 1997. Vol.1.

[RS97b] Rozenberg, G., Salomaa, A., (eds.): *Handbook of Formal Languages.* Springer, Berlin 1997. Vol.2.

[S00] Skryja, J.: *Semihomogenous Finite Automata.* Diploma thesis, Dept. of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, January 2000.

Crystal Lattice Automata

Jim Morey, Kamran Sedig, Robert E. Mercer, and Wayne Wilson

Cognitive Engineering Lab, University of Western Ontario
{morey,mercer,wawilson}@csd.uwo.ca, sedig@uwo.ca

Abstract. A description of crystal lattices in terms of automata is presented. The words of a language represented by an automata are mapped to points in $\mathbf{R}^2$ and $\mathbf{R}^3$ defining lattice points and their connections. These automata descriptions of crystal lattices reveal subtle properties that are difficult to see in other descriptions. A few applications of these automata are discussed.

Keywords: automata, crystallography, lattice, tiling, microworlds

1 Introduction

Two and three-dimensional lattice structures have been studied in art, mathematics, and chemistry. The lattice structures share the quality of being complex, infinite, and yet follow some repeating pattern. The regularity in the lattice permits compact representation. We introduce the crystal lattice model, an abstract model of a lattice with the following properties: there is a rigid arrangement of an infinite number of nodes; each node has a well-defined local geometry that describes the node's connections to nearby nodes; and the arrangement has no limit points (limit points are not physically realistic). This abstraction provides the platform for our research.

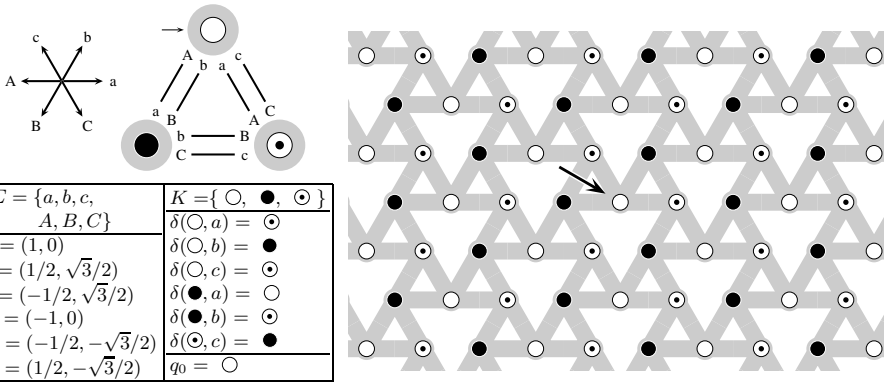
2 A Crystal Lattice Automaton

Consider a bi-directional automaton, an automaton that has only bi-directional transitions. Explicitly, $M_{bi} = (K, \Sigma, \delta, K, q_0)$ where K is a finite set of states (all states are final), Σ is a finite alphabet, δ is a map from $K \times \Sigma$ into K, and $q_0 \in K$ is the start state. The transitions are bi-directional: $\forall a \in \Sigma \; \exists! A \in \Sigma, q, q' \in K$ such that if $\delta(q, a) = q'$ then $\delta(q', A) = q$. A is refered to as the inverse of a. For simplicity in diagrams, a letter and it's inverse are denoted by a small letter and the corresponding capital letter.

The language of the M_{bi}, $L_{M_{bi}}$, is mapped into $\mathbf{R}^2$ and $\mathbf{R}^3$ by the map p, short for position. For the letter $a \in \Sigma$, $p(a) = r_a, r_a \in \mathbf{R}^2$ or $\mathbf{R}^3$; for the inverse A, $p(A) = -r_a$; for the word $w = (a_0 a_1 a_2 \cdots a_n) \in L_{M_{bi}}$, $a_i \in \Sigma$, $p(w) = \sum_{i=0}^{n} p(a_i)$.

A connected structure, we call a lattice, can be produced by p and M_{bi} with the introduction of the map c that maps a word from $L_{M_{bi}}$ into a ordered pair

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 214–220, 2002.
© Springer-Verlag Berlin Heidelberg 2002



$\Sigma = \{a, b, c, A, B, C\}$	$K = \{ \bigcirc, \bullet, \odot \}$
$a = (1, 0)$	$\delta(\bigcirc, a) = \odot$
$b = (1/2, \sqrt{3}/2)$	$\delta(\bigcirc, b) = \bullet$
$c = (-1/2, \sqrt{3}/2)$	$\delta(\bigcirc, c) = \odot$
$A = (-1, 0)$	$\delta(\bullet, a) = \bigcirc$
$B = (-1/2, -\sqrt{3}/2)$	$\delta(\bullet, b) = \odot$
$C = (1/2, -\sqrt{3}/2)$	$\delta(\odot, c) = \bullet$
	$q_0 = \bigcirc$

Fig. 1. Two representations of a CLA along with the associated lattice

that represents the position of a node and a set of positions of connected nodes, $c(w) = (p(w), \{p(wa) \mid wa \in L_{M_{bi}}, a \in \Sigma\})$. The lattice, $l(M_{bi}, p)$, is defined by the set of all connections, $\{c(w) \mid w \in L_{M_{bi}}\}$. The notion of mapping regular languages to mathematical structures can also be found in Epstein's construction of automatic groups, where regular languages are mapped into groups [1]. Crystal lattices have the restriction that there are no limit points *(P1)* and the restriction that the connections are well defined *(P2)*.

$$\inf(\{\|p(w) - p(w')\| \mid w, w' \in L_{M_bi}\} \setminus 0\}) > 0 \qquad (P1)$$

$$\forall w, w' \in L_{M_{bi}} p(w) = p(w') \rightarrow c(w) = c(w') \qquad (P2)$$

If $l(M_{bi}, p)$ has *P1* and *P2*, then $l(M_{bi}, p)$ is a **crystal lattice** and we will call the pair (M_{bi}, p) a **crystal lattice automaton** (CLA).

In Figure 1 the diagrammatic representation of the CLA uses two conventions. First, the direction letters near a state represent accepted transitions from the state. The local geometry of a state can be determined by a set of direction letters. For instance, the white state has the local geometry of a "K" on its side created by the directions a, b, c, and A. Second, the outgoing transitions are also the incoming transitions of another state. To promote the bi-directional relationship of CLA transitions the two transitions are unified into one edge with a direction letter on each end that represents the two transitions.

In Figure 2 we see a visual representation of the mapping into the plane of words associated with the CLA in Figure 1. All the words accepted by the CLA are mapped to nodes of the lattice. Although many words are mapped to the same position there is no problem with ill-defined local geometries at a position because *P2* insures that at a particular position the local geometries of different words are the same.

The properties of the language associated with CLA can reveal some of the properties of the lattice. For example, the language associated with Figure 1 has no words with substrings that are of the form x^3, $x \in \Sigma$. This property reveals that the lattice has no straight segments longer than 2. Another property

locations	accepted words
$(0, 0)$	ϵ, aA, bB,cC, Aa, $\cdots$
$(1, 0)$	a, bC, acC, $\cdots$
$(\frac{1}{2}, \frac{\sqrt{3}}{2})$	b, ac, baA, $\cdots$
$(-\frac{1}{2}, \frac{\sqrt{3}}{2})$	c, Ac, ccC, $\cdots$
$(-1, 0)$	A, cB, aAA, $\cdots$
$(\frac{1}{2}, -\frac{\sqrt{3}}{2})$	aB, aCA, $\cdots$
$(1\frac{1}{2}, -\frac{\sqrt{3}}{2})$	aC, aBa, $\cdots$
$(1\frac{1}{2}, \frac{\sqrt{3}}{2})$	ba, bbC, $\cdots$
$(1, \sqrt{3})$	bb, bac, $\cdots$
$(-1, \sqrt{3})$	cc, cAb, $\cdots$
$(-1\frac{1}{2}, \frac{\sqrt{3}}{2})$	cA, ccB, $\cdots$
$(-1\frac{1}{2}, -\frac{\sqrt{3}}{2})$	AB, ACA, $\cdots$
$(-\frac{1}{2}, -\frac{\sqrt{3}}{2})$	AC, ABa, $\cdots$
$(0, \sqrt{3})$	aBB, ACC, $\cdots$
$\vdots$	$\vdots$

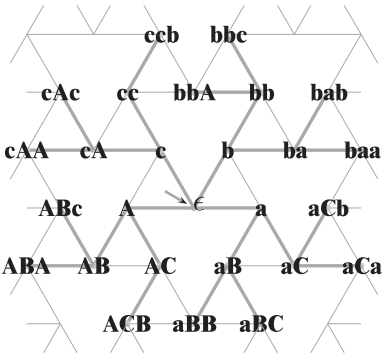


Fig. 2. A list of accepted words and their corresponding locations

of the language is that certain permutations on the letters leave the language unchanged. These permutations correspond to symmetries in the lattices. For instance, the permutation of letters, $(aA)(cb)(BC)$, reveals the line of symmetry in the lattice, $(x, y) = (-x, y)$.

3 Lattice Properties

Although the CLA definition admits many types of lattices, only certain types may be of interest. One type of lattice that is geometrically interesting is a lattice that has only one type of local geometry, this property we refer to as **regular**. For instance, the lattice in Figure 1 has the "K" geometry at each node (possibly rotated). Creating a number of regular lattices with a particular local geometry can be difficult since this further restriction of the CLA limits many choices. If a lattice is not regular, then it is **n-regular**; the lattice has exactly n different local geometries. It is clear that given a finite alphabet of directions, Σ, the number of possible local geometries is finite since it must be bounded above by $2^{|\Sigma|}$. Figure 3 shows a *2-regular* lattice with the local geometries that look like a "y" and an "ϵ". The outer states are the "y" states and the inner states, with the lines through them, are the "ϵ" states. Notice that the state symbols, the circles, correspond to the symbols of the original CLA shown in Figure 1.

Another important property that helps organize lattices, is the concept of sub-lattices. A lattice, S, is a **sub-lattice** of L if all of the nodes of S are in L and the connections of S are a strict subset of the connections of L. Figure 3 shows a sub-lattice of the lattice in Figure 1. The original lattice is shown in dotted lines. Along with the idea of sub-lattice comes the idea of a **full** lattice.

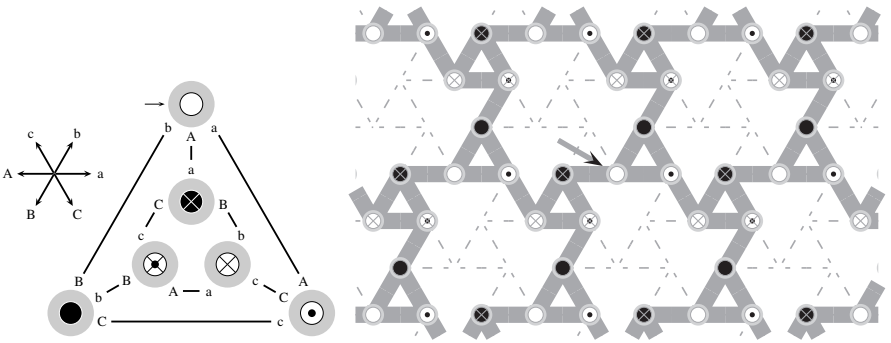


Fig. 3. a 2-regular sub-lattice of the lattice in Figure 1

A full lattice has all of the words in Σ^*. An example of a full lattice would be the triangular tessellation of the plane. Notice that the lattices in Figures 1 and 3 are both sub-lattices of this full lattice. However, there is no full lattice for the $\Sigma = \{(\cos(k * \Pi/5), \sin(k * \Pi/5)), k = 0...10\}$ since this would contain limit points, violating *P1*.

Finally, one further property that all of the previous lattices share is that all of the connections between the nodes are of unit length. We refer to this property as being **uniform**. Thus the lattice in Figure 3 can be referred to as a *2-regular uniform* lattice. **N-uniform** refers to lattices that contain n different lengths of connections.

4 Applications

There are three main areas in which we are currently applying CLAs: in two dimensional geometry, in three dimensional navigation, and in crystallography. The CLAs are intended to be cognitive artifacts [4] that aid in understanding the particular subject.

In the geometrical direction, an educational environment called "K Lattice World" [7] (Figure 4) was developed around regular lattices that have 'k'-like local geometries. This world allows learners to explore the connections between the automata and the lattices through a series of activities.

In the navigation area, three dimensional regular lattices created by CLAs have qualities that can be used in navigation research. For example, getting "lost" in one of these lattices is not difficult, since the local geometry everywhere is exactly the same and yet there may be neighbourhoods that are quite distinguishable. Certain inherent landmarks come from the automata while others can be imposed. For instance, in Figure 5 the lattice has been truncated so that it is no longer infinite. The edges of the lattice become landmarks that aid in navigation. LatticeSpace is an environment that allows learners to explore lattices. One of the exploration techniques uses node labeling to help construct landmarks that aid in conceptualizing the lattice. LatticeSpace currently focusses

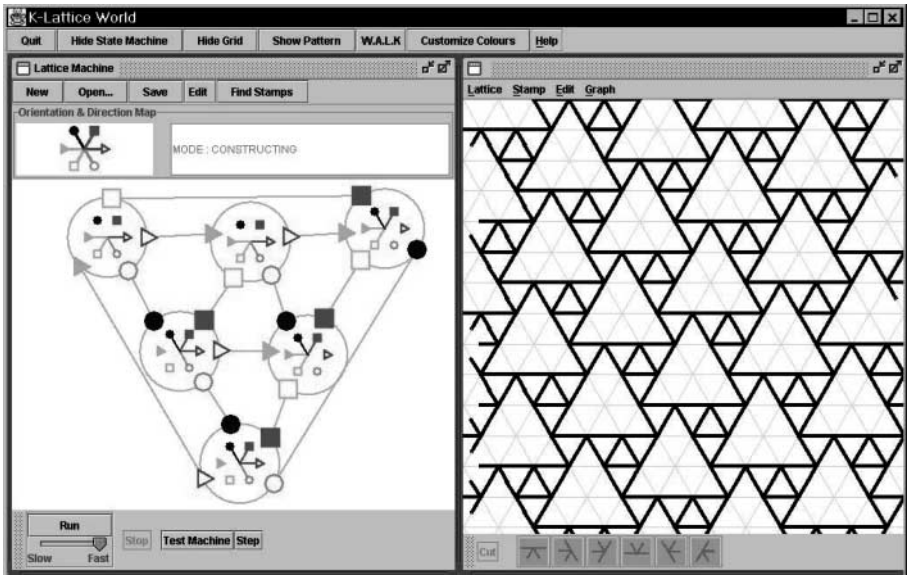


Fig. 4. A screen capture of K Lattice World

on regular sub-lattices of well known full lattices (for instance, the full lattice with local geometry $\{(\pm 1, 0, 0), (\pm 1, 0, 0), (\pm 1, 0, 0)\}$).

In the crystallography area, the CLA description of lattices allows for a different representation of crystals. In chemistry, crystal lattices are described using a tiling representation [2, 5]. The tiles, called unit cells, are often parallelepipeds. A unit cell's description contains the cells lengths, the angles between its edges, and the positions of all the atoms contained in the cell. A unit cell's description can be quite long as it may contain many atoms. This description can be greatly condensed using one of the 230 space groups [3]. In the above representation, the unit cells are described explicitly, and the atoms and their bonds are described implicitly [5]. The local geometry of many of the atoms and bonds can be difficult to determine using the above representation. The crystal lattices generated by the CLA describe the atoms and their bonds explicitly and the unit cells implicitly. Figure 6 highlights the main difference between the standard tiling and the automaton forms of representation of lattices. In the tiling representation, the segment of the lattice shown in the figure is what would be considered a unit cell, where the parallelepiped is a cube. Using the tiling form of representation, one is concerned about the unit cell and its internals [3, 5]. However, in the automaton representation, one can focus directly on the atoms (nodes) and bonds (connections) making up the lattice. Although informationally equivalent, the two representations are not equivalent in terms of computational processing and how they lend themselves to different tasks and activities [6].

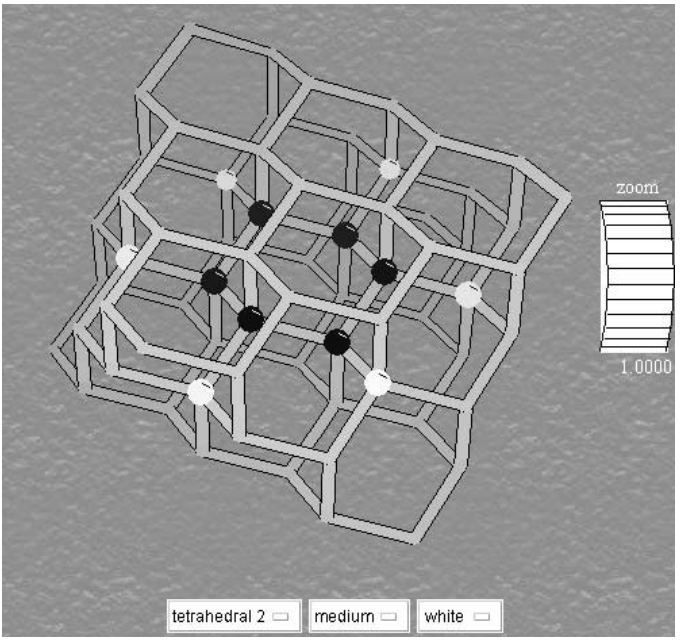


Fig. 5. A screen capture of LatticeSpace

5 Conclusions

In this paper, we have presented a description of crystal lattices in terms of automata. We have used a few simple lattice structures to demonstrate how to represent lattices using automata. The focus on the simple local geometry of regular and uniform lattices (e.g., 'K's used in the K-Lattice World) has allowed us to examine implicit lattice automata properties. In addition to improving our existing microworlds and geometric games, we are exploring alternative automata representations for crystal lattices.

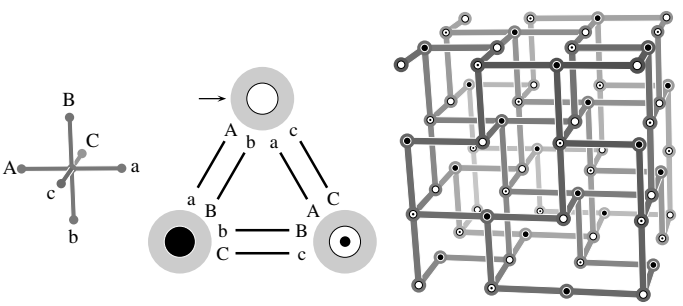


Fig. 6. A 3D lattice (based on the automaton in Figure 1)

Acknowledgement

This research is supported by NSERC funding.

References

[1] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston. *Word Processing in Groups*. Jones and Barlett Publishers, Boston, London, 1992. 215

[2] Branko GrunBaum and G. C. Shephard. *Tilings and Patterns: An Introduction*. W. H. Freeman and Company, New York, 1989. 218

[3] Thomas C. W. Mak and Gong-Du Zhou. *Crystallography in Modern Chemistry: A Resource Book of Crystal Structures*. John Wiley & Sons, Inc., New York, 1992. 218

[4] D. A. Norman. *Things that Make Us Smart: Defending Human Attributes in the Age of the Mahine*. Addison-Wesley Publishing Company, Reading MA, 1993. 217

[5] David W. Oxtoby and Norman H. Nachtrieb. *Principles of Modern Chemistry*. Saunders College Publishing, New York, 1986. 218

[6] Donald Peterson, editor. *Forms of Representation*. Intellect, Exeter, UK, 1996. 218

[7] Wayne A. Wilson. K-lattice world: A tool to explore interactive representations of lattice patterns. Master's thesis, University of Western Ontario, 2001. 217

Minimal Adaptive Pattern-Matching Automata for Efficient Term Rewriting

Nadia Nedjah and Luiza de Macedo Mourelle

Department of Systems Engineering and Computation, Faculty of Engineering
State University of Rio de Janeiro
Rio de Janeiro, Brazil
{nadia,ldmm}@eng.uerj.br
http://www.eng.uerj.br/~ldmm

Abstract. In term rewriting systems, pattern-matching is performed according to a prescribed traversal order. By adapting the traversal order to suit the patterns of the rewriting system, it is often possible to obtain *better* matching automata in the sense that they have a smaller size and allow term matching in shorter time, compared with the left-to-right automaton. They may improve termination properties too. Space requirement is reduced further using a directed acyclic graph automaton that shares all the equivalent subautomata. This is done without altering either the matching time or termination properties. Isomorphic subautomata are duplicated in the tree-based automata. We discuss and develop an efficient approach to space requirement optimisation, matching time and termination properties improvement.

1 Introduction

Pattern matching of terms using the left-to-right tree automaton identifies a match (or lack thereof) after a single scan of the target term. Unnecessary positions may be inspected to ensure that no backtracking is needed when pattern matching fails [9, 5]. Using the left-to-right automaton with the adaptive strategy [1, 3, 4, 7, 13], the evaluation of the subterms rooted at inspected positions is forced in the left-to-right order. If such evaluations do not terminate, the evaluation of the subject term does not terminate either [2, 6, 8, 16]. However, the order of pattern matching terms need not be left-to-right. Sometimes, the evaluation of the same subject term would terminate if subterms were evaluated in a different order so that non-terminating evaluations might be avoided. We illustrate this through the following example:

Example 1. Consider the equational program with the following set of rewrite rules:

$$\begin{array}{lll} f(a, \omega, a) & \rightarrow a & (r_1) \\ f(a, b, b) & \rightarrow b & (r_2) \\ c & \rightarrow c & (r_3) \end{array}$$

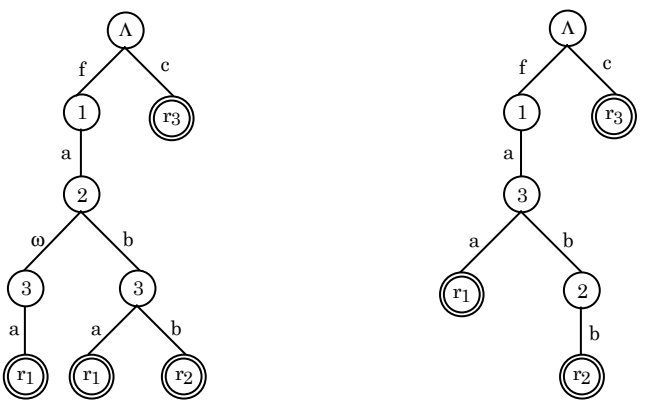
B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 221-233, 2002.
© Springer-Verlag Berlin Heidelberg 2002

where $\#f = 3$ and as usual a, b and c are constants. Consider the matching automata of Fig. 1 where non-final states are labelled inside with the positions inspected and final states are labelled with the name of the matched rule.

With appropriate data structures, pattern-matching can be performed in any given order. So, one way to improve efficiency i.e., matching times, which may improve termination too, consists of adapting the traversal order to suit the input patterns. We identify such an adaptive traversal order for a given pattern set and construct the corresponding automaton.

We first introduce from [12] a method that for any given traversal order constructs the corresponding adaptive matching automaton. *Indexes* are positions whose inspection is necessary to declare a match. Inspecting them first for patterns which are not essentially strongly sequential [9] allows us to engineer adaptive traversal orders that should improve space usage and matching times as shown in [12, 14]. When no index can be found for a pattern set, we showed [9] that a position, which is an index for a maximal number of high priority patterns can always be identified. Selecting such a position attempts to improve matching times for terms that match patterns of high priority. A *good* traversal order, one which improves space, time and termination properties [12], inspects positions that are indexes/partial indexes for a pattern set.

In this paper, space requirements of matching automata are further reduced by using a directed acyclic graph (*dag*) automaton that shares all the isomorphic subautomata, which are duplicated in the tree automaton. We design an efficient method to identify such subautomata and avoid duplicating their construction while generating the dag automaton. We generalise some results of [11] to be able to construct adaptive dag automata directly. Then, we compare left-to-right matching automata to obtained adaptive automata.



(a) *Left-to-right automaton* **(b)** *Adaptive automaton*

Fig. 1. Matching automata for {r_1: faωa, r_2: fabb, r_3: c}

2 Notation and Definitions

In the rest of the paper, we will use some notation and concepts defined as follows: symbols in a *term* are either function or variable symbols; the non-empty set of function symbols that appear in the patterns $F = \{a, b, f, g, h, ...\}$ is *ranked* i.e., every function symbol f in F has an *arity* which is the number of its arguments and is denoted $\#f$; a term is either a constant, a variable or has the form $ft_1\, t_2 ... t_{\#f}$ where each t_i, $1 \le i \le \#f$, is itself a term. We abbreviate terms by removing the usual parentheses and commas. This is unambiguous in our examples since the function arities will be kept unchanged throughout, namely $\#f = 3$, $\#g = 1$, $\#h = 1$, $\#a = \#b = 0$. Variable occurrences are replaced by ω, a meta-symbol, which is used since the actual symbols are irrelevant here. A term containing no variables is said to be a *ground* term. We generally assume that patterns are linear terms, i.e. each variable symbol can occur at most once in them. Pattern sets will be denoted by L and patterns by π_1, π_2, ..., or simply by π. A term t is said to be an *instance* of a (linear) pattern π if t can be obtained from π by replacing the variables of π by corresponding subterms of t. If term t is an instance of pattern π then we denote this by $t \lhd \pi$.

Here, we assume that we are free to choose any order for pattern-matching terms and their evaluation proceeds using the adaptive strategy. In particular, if $f(t_1, ..., t_n)$ is the term being matched at the root, the argument t_n may be reduced before the argument t_j, $j \le i$ if the pattern-matcher visits the position at which t_n is rooted before that at which t_j is rooted. We will say that a term t *matches* a pattern $\pi \in L$ if, and only if, t is an instance of π, i.e. $t \lhd \pi$ and t is not an instance of any other pattern in L, i.e. higher priority than π. A *position* in a term is a path specification, which identifies a node in the parse tree of the term. Position is specified here using a list of positive integers. The empty list Λ denotes the position of the root of the parse tree and the position $p.k$ ($k \ge 1$) denotes the root of the kth. argument of the function symbol at position p.

A *matching item* is a pattern in which all the symbols already matched are now *ticked* i.e., they have the *check-mark* ✓ Moreover, it contains the *matching dot* • which only designates the *matching symbol* i.e., the symbol to be accepted next. The position of the matching symbol is called the *matching position*. A *final* matching item, namely one of the form π•, has the *final* matching position, which we write ∞. Final matching item may contain unchecked positions. These positions are irrelevant for announcing a match and so must be labelled with the symbol ω. Matching items are associated with a rule name.

The term obtained from a given item by replacing all the terms with an *unticked* root symbol by the placeholder _ is called the *context* of the items. For instance, the context of the item f✓a•$g\omega a$✓ is the term $f(_, _, a)$ where the arities of f, g and a are as usual 3, 2 and 0. In fact, no symbol will be checked until all its parents are all checked. So, the positions of the placeholders in the context of an item are the positions of the subterms that have not been checked yet. The set of such positions for an item i is denoted by $up(i)$ (short for unchecked positions).

A *matching set* is a set of matching items that have the same context and a common matching position. The *initial* matching set contains items of the form •π

because we recognise the root symbol (which occurs first) first whereas, *final* matching sets contain items of the form $\pi\bullet$, where π is a pattern. For initial matching sets, no symbol is ticked. A final matching set must contain a final matching item i.e., in which all the unticked symbols are ωs. Furthermore, the rule associated with that item must be of highest priority amongst the items in the matching set.

If $L\cup\{\pi\}$ is a prioritised pattern set, then π is said to be *relevant for L* if there is a term that matches π in $L\cup\{\pi\}$. Otherwise, π is *irrelevant for L*. Similarly, an item π is *relevant for* (*the matching set*) *M* if there is a term that *deterministically* matches π in $M\cup\{\pi\}$. Since the items in a matching set *M* have a common context, they all share a common list of unchecked positions and we can safely write *up*(*M*). The only unchecked position for an initial matching set is clearly the empty position Λ.

3 Adaptive Tree Matching Automata

States of adaptive tree automata are labelled by matching sets. Since here the traversal order is not fixed a priori (i.e., it will be computed during the automaton construction procedure), the symbols in the patterns may be accepted in any order. When the adaptive order coincides with the left-to-right order, matching items and matching sets coincide with matching items and matching set respectively [9].

We describe adaptive automata by a 4-tuple $\langle S_0, S, Q, \delta\rangle$. *S* is the state set, $S_0\in S$ is the initial state, $Q\subseteq S$ is the set of final states and δ is the state transition function. The states are labelled by matching sets, which consist of original patterns together with extra instances of the patterns, which are added to avoid backtracking in reading the input. In particular, the matching set for S_0 contains the initial matching items formed from the original patterns and labelled by the rules associated with them. The transition function δ of an adaptive automaton is defined using three functions namely, *accept*, *choose*, and *close*: $\delta(M, s) = close(accept(M, s, choose(M, s)))$. The function *accept* and *close* are similar to those of the same name in [12], and *choose* picks the next matching position. For each of these functions, we give an informal description followed by a formal definition, except for *choose* for which we present an informal description. Its formal definition will be discussed in detail in the next section.

For a matching set and an unchecked position, function *accept* accepts and ticks the symbol immediately after the matching dot in the items of the matching set and inserts the matching dot immediately before the symbol at the given unchecked position. Let *t* be a term in which some symbols are ticked. We denote by $t_{\bullet p}$ the matching item which is *t* with the matching dot inserted immediately before the symbol $t[p]$. Then the definition of *accept* is $accept(M, s, p) = \{(\alpha s\checkmark\beta)_{\bullet p} | \alpha\bullet s\beta\in M\}$.

Function *choose* selects a position among those that are unchecked for the matching set obtained after accepting the symbol given. For a matching set *M* and a symbol *s*, the *choose* function selects the position which should be inspected next. The function may also return the final position ∞. In general, the set of such positions is denoted by $\overline{up}(M, s)$. It represents the unchecked positions of $\delta(M, s)$, and consists of $up(M)$ with the position of the symbol *s* removed. Moreover, if the arity of *s* is

positive then there are #s additional unchecked terms in the items of the set $\delta(M, s)$, assuming that ωs are of arity 0. Therefore, the positions of these terms are added. Recall that if a function symbol f is at position p in a term, then the arguments of f occur at positions $p.1, p.2, \ldots, p.\#f$. We can now formulate a definition for the set of unchecked positions $up(\delta(M, s)) = \overline{up}(M, s)$ as follows, wherein p_s is the position of s which is also the matching position of M.

$$\overline{up}(M,s) = \begin{cases} up(M) \setminus \{p_s\} & \text{if } \#s = 0 \\ (up(M) \setminus \{p_s\}) \cup \{p_s.1, \ldots, p_s.\#s\} & \text{otherwise} \end{cases}$$

Given a matching set, function *close* computes its closure in the same way as for the left-to-right matching automaton. As it is shown in the following, the function adds an item $\alpha \bullet f\omega^{\#f}\beta$ to the given matching set M whenever an item $\alpha \bullet \omega\beta$ is in M together with at least one item of the form $\alpha \bullet f\beta'$. The definition of function *close* allows us to avoid introducing irrelevant items [12] wherein $\prec$ expresses the priority rule.

$$close(M) = M \cup \{\, r\colon \alpha \bullet f\omega^{\#f}\beta \mid r\colon \alpha \bullet \omega\beta \in M \text{ and } \exists \beta' \text{ s.t. } r'\colon \alpha \bullet f\beta' \in M \text{ and } \forall \beta'' \text{ If } r''\colon \alpha \bullet f\beta'' \in M \text{ and } \alpha\omega\beta \prec \alpha f\beta'' \text{ Then } \neg(\alpha f\omega^{\#f}\beta \lhd \alpha f\beta'')\}$$

For a matching set M and a symbol $s \in F \cup \{\omega\}$, the transition function for an adaptive automaton can now be formally defined by the composition of the three functions accept, choose and close is $\delta(M, s) = close(accept(M, s, choose(M, s)))$. This function proceeds by first computing the list of unchecked positions of the expected set $\delta(M, s)$, then selecting an unchecked position p in $\overline{up}(M,s)$ and computing the matching set $accept(M, s, p)$ and finally computing its closure of N. The matching set $accept(M, s, p)$ is called the *kernel* of $\delta(M, s)$.

It is clear that a *good* traversal order should select *indexes* first whenever possible. When no index can be found for a matching set, a *partial index* can be selected. If more than one partial index is available then some heuristics can be used to discriminate between them (for more details see [9, 12]).

Example 2. Consider the set L={1:$hgga\omega\omega a$, 2:$hgfa\omega\omega aa$, 3:$h\omega b$}. An adaptive automaton for L using matching sets is shown in Fig. 2. The choice of the positions to inspect will be explained later. Transitions corresponding to failures are omitted, and the ω-transition is only taken when there is no other available transition which accepts the current symbol. Notice that for each of the matching set in the automaton, the items have a common set of symbols checked. Accepting the function symbol h from the initial matching set (i.e., that which labels state 0) yields the set {1:$h\checkmark \bullet gga\omega\omega a$, 2:$h\checkmark \bullet gfa\omega\omega aa$, 3:$h\checkmark \bullet \omega b$, 3:$h\checkmark \bullet g\omega\omega b$} if position 1 is to be chosen next. Then, the closure function adds the item $h\checkmark \bullet g\omega\omega b$. Provided with a matching set M, the *choose* function selects positions that are labelled with a function symbol in at least one item of M. If more than one position is available then the leftmost is selected. In short, the traversal order used avoids positions labelled with ω in every item of M.

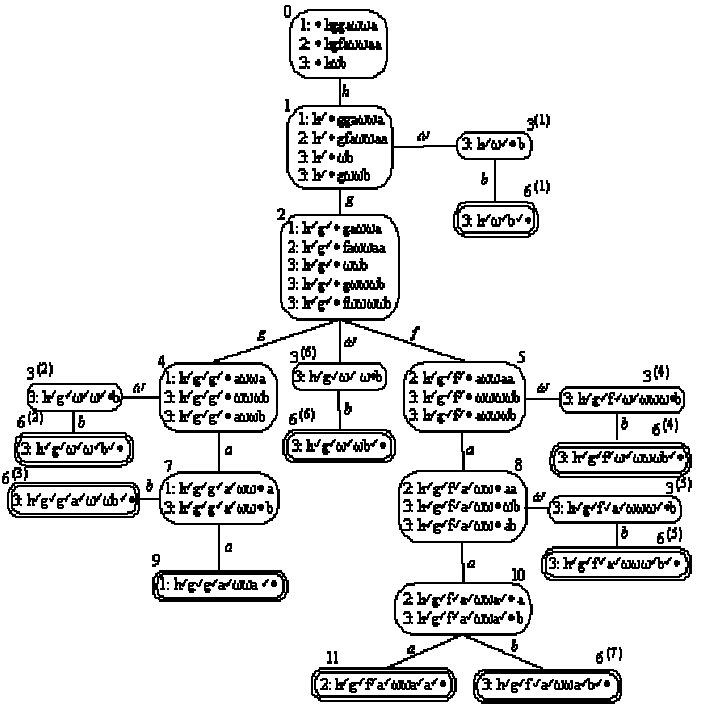


Fig. 2. An adaptive tree automaton for {1:hggaωωa, 2:hgfaωωaa, 3:hωb}

4 Optimal Adaptive Matching Automata

The tree automaton described above is time efficient during operation because it avoids symbol re-examination. However, it achieves this at the cost of increased space requirements. The unexpanded automaton corresponding to the pattern set of Fig. 2, and to which no patterns are added, is given in Fig. 3. In that automaton, states are also labelled with the matching position. For instance, state 0 investigates position Λ and states 4 and 5 both scans position 1.1.1. Furthermore, final state are also labelled with the number of matched rule. The non-deterministic automaton is much smaller. For instance, there $hg\omega\omega b$ is only recognised by backtracking from state 2 to state 1 and then taking the branch through state 3 instead. But in Fig. 2 a branch recognising $hg\omega\omega b$ has been added to avoid backtracking, thereby duplicating the existing sub-branch which recognises the b in $h\omega b$. We can see similar duplication in several other branches of Fig. 2; those identified by sharing the same main state numbers.

By sharing duplicated branches, tree automata can be converted into an equivalent but smaller directed acyclic graph (*dag*) automata. States, which recognise the same inputs and assign the same rule numbers to them are functionally equivalent, and can

be identified. For instance, the dag automaton corresponding to Fig. 2 is given in Fig. 4. The state number is thereby reduced from 22 to 12.

In tree-based adaptive matching automata, functionally identical (or isomorphic) subautomata may be duplicated. Using directed acyclic graphs instead of trees can then improve the size of adaptive automata. In this section, we define the notion of matching item equivalence so that a dag-based adaptive automata can be generated.

The illustrative example hides the complexity of recognising duplication where a number of suffixes are being recognised, not just one. The required dag automaton can be generated using finite state automaton minimisation techniques but this may require a lot of memory and time. The obvious alternative approach consists of using the matching sets to check new states for equality with existing ones while generating the automaton. In the case of equality, the new state is discarded and the existing one is shared. However, comparison of matching sets may be prohibitively expensive and it may well require bookkeeping for all previously generated matching sets. A major aim of this paper is to show how to avoid much of this work. First, we must characterise states that would generate isomorphic subautomata.

Definition 1. Let $i_1 = r_1{:}\alpha_1 \bullet \beta_1$ and $i_2 = r_2{:}\alpha_2 \bullet \beta_2$ be two matching items and p a position in $up(i_1) \cup up(i_2)$. i_1 and i_2 are *equivalent* if, and only if, $r_1 = r_2$ and the symbols labelling unchecked positions in i_1 and i_2 are as follows, otherwise, i_1 and i_2 are *different*.

$$\begin{cases} \alpha_1\beta_1[p] = \alpha_2\beta_{21}[p] & \text{if } p \in up(i_1) \cap up(i_2) \\ \alpha_1\beta_1[p] = \omega & \text{if } p \in up(i_1) \setminus up(i_2) \\ \alpha_2\beta_2[p] = \omega & \text{if } p \in up(i_2) \setminus up(i_1) \end{cases}$$

Definition 2. Two matching sets M_1 and M_2 are *equivalent* if, and only if, to every item i in $M_1 \cup M_2$ there correspond items $i_1 \in M_1$ and $i_2 \in M_2$ which are equivalent to i. Otherwise, the sets are *different*.

For instance, in the adaptive matching automaton of Fig. 2, the matching sets labelling the states $3^{(1)}$ and $3^{(2)}$ are equivalent. So, if two matching sets are equivalent, then their items differ only in those unchecked positions that occur in either of the matching sets. Moreover, for every item in which such positions occur, they are labelled with the symbol ω and so irrelevant for declaring a match. Therefore, equivalence a s formalised in Definitions 1 and 2 is the right criterion for coalescing nodes of the adaptive tree automaton to obtain the equivalent adaptive dag automaton. It is clear that two matching sets generate identical automata if they are equivalent.

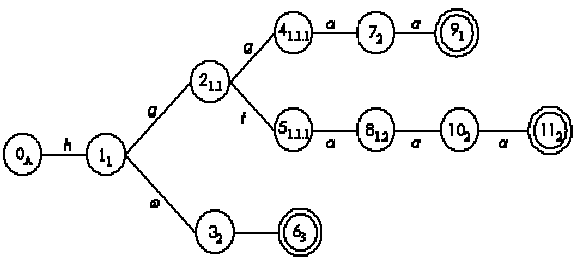


Fig. 3. Unexpanded adaptive tree automaton for {1:hggaωωa, 2:hgfaωωaa, 3:hωb}

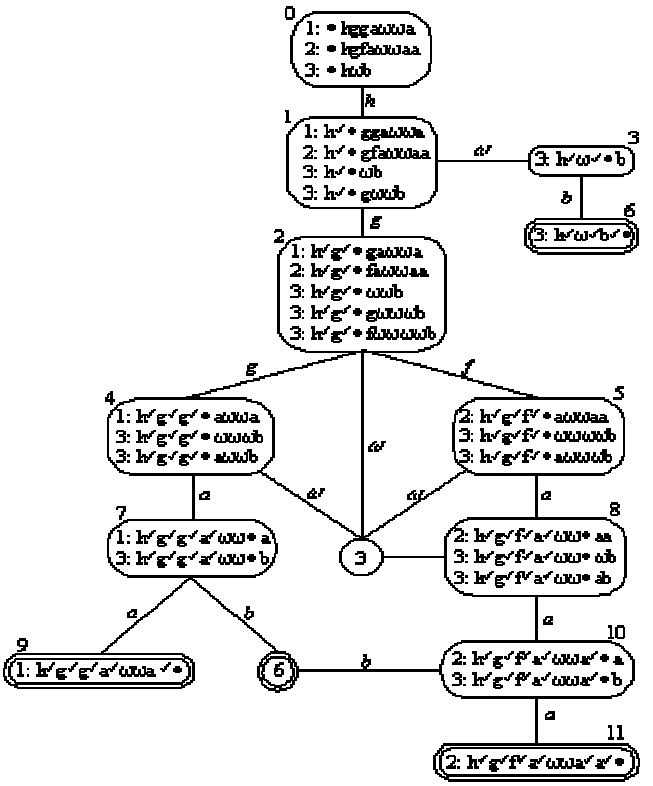


Fig. 4. Adaptive dag automaton for {1:hggaωωa, 2:hgfaωωaa, 3:hωb}

For equivalent sets, using the same traversal order strategy, function *choose* will certainly select the same positions. They must occur in the intersection of their unchecked position sets. So equivalent matching sets generate equivalent adaptive automata. We believe this equivalence is actually necessary as well as sufficient to combine corresponding states in the automaton. Equivalent matching sets may have different contexts, as can be seen in Fig. 2.

5 Adaptive Dag Automata Construction

In this section, we describe how to build the minimised dag automaton efficiently without constructing the tree automaton first. This requires the construction of a list of matching sets in a suitable order to ensure that every possible state is obtained, and a means of identifying potentially equivalent states.

The items in matching sets all share a common context. Hence, the matching position of any item is an invariant of the whole matching set. The states of the tree automaton can therefore be ordered using the left-to-right total ordering on the common matching positions of their matching sets. The dag automaton is constructed

as in Algorithm 1. We iteratively construct the machine using a list l of matching sets in which the sets are ordered according to the matching position of the set. So the initial matching set is first and final matching sets come last. Each set in l is paired with its state in the automaton and a pointer is kept to the current position in l.

The list l represents the equivalence classes of matching states where the assigned matching position is that of the representative which is generated first. In each new set which is generated, the position of the current matching set is incremented at least one place to the right. So new members of l are always inserted to the right of the current position. This ensures that all necessary transitions will eventually be generated without moving the pointer backwards in l. It is easy to see from the definition of the *close* function that added patterns cannot contain positions that were not in one of the original patterns. So l only contains sets with matching positions from a finite collection and, as each set can only generate a finite number of next states all of which are to the right, the total length of l is bounded and the algorithm must terminate.

The list l represents the equivalence classes of matching states where the assigned matching position is that of the representative which is generated first. New members of l are always inserted to the right of the current position. This ensures that all necessary transitions will eventually be generated without moving the pointer backwards in l. It is easy to see from the definition of the *close* function that added patterns cannot contain positions that were not in one of the original patterns. So l only contains sets with matching positions from a finite collection and, as each set can only generate a finite number of next states, the total length of l is bounded and the algorithm must terminate. Furthermore, the tree and dag automata clearly accept the same language and the automaton is minimal, in the sense that, by construction, none of the matching sets labelling the states in the automaton are equivalent.

algorithm *ConstructDagAutomaton*(pattern set L);
    A $\leftarrow \emptyset$;
    $l \leftarrow \{\langle M_0 \leftarrow (\forall \pi \in L, \bullet\pi), S_0\rangle\}$;
    current $\leftarrow 0$;
    do
        for each $s \in F \cup \{\omega\}$ **do**
            compute $\delta(M, s)$;
            if $\nexists\ \langle M', S'\rangle \in l \mid \delta(M,s) \equiv M'$ **then**
                create state S' labelled with $\delta(M,s)$;
                add transition S $\xrightarrow{s}$ S' to A;
                add pair $\langle \delta(M,s), S'\rangle$ to l;
            else
                add transition S $\xrightarrow{s}$ S' to A;
        current $\leftarrow$ current + 1;
    while current $\neq$ null;
end algorithm;

Algorithm 1. Constructing Adaptive Dag Automata

6 State Equivalence and Automata Complexity

In this section, we show how matching sets can frequently be discriminated easily so that the cost of checking for equivalence is minimised. Comparison of suffixes in the matching sets is completely avoided. We look at two properties to help achieve this. One is the set of rules represented by patterns in the matching set and the other is the matching position.

In Fig. 4, all different matching sets are distinguished by the use of the rule set. Thus the criterion is useful in practice. However, it will clearly not be sufficient in general. In the opposite direction, it is also sometimes easy to establish equivalence. Combining it with the matching position, we have the following very useful result, which enables the direct checking of equivalence to be avoided entirely in Example 2.

Theorem 1. Matching sets that share a common matching position and rule set are equivalent.

Proof. It suffices to show the kernels of the matching sets are equivalent, since then the function *close* will add equivalent items to both sets. Let M_1 and M_2 be two matching sets that share the same rule set and common matching position p. Let $i_1 = r{:}\alpha_1 \bullet \beta_1$ and $i_2 = r{:}\alpha_2 \bullet \beta_2$ be any two items associated with the same rule in their respectively kernels. The definitions of *accept* and *close* guarantee that the suffixes consist of a suffix of the original pattern π_r of the rule preceded by a number of copies of ω. To identify this suffix, let p' be the maximal prefix of the position p corresponding to a symbol in π_r. This is either the whole of p or is the position of a variable symbol ω. Either ways, substitutions made by *close* for variables before p' in either i_1 or i_2 have already been fully passed in the prefix, and no substitution has yet been made for any variable further on in π_r. So if β is the suffix of π_r that starts at p' then the items i_1 and i_2 must have the form $\alpha_1 \bullet \omega^{n_1}\beta$ and $\alpha_2 \bullet \omega^{n_2}\beta$ for some $n_1, n_2 \geq 0$. It is clear that those items are equivalent in the sense of Definition 2. We conclude that M_1 and M_2 are equivalent. ■

We now concentrate on evaluating the space complexity of the adaptive dag automaton by giving an upper bound for its size in terms of the number of patterns and symbols in the original pattern set. The bound established considerably improves left-to-right dag automata bound [5, 10]. The size of the left-to-right dag automaton for a pattern set L is bounded above by: $1+|L|+(2^{|\pi|}-1)(\ \Sigma_{\pi \in L}(|\pi|-1)\ )$ (see [10] for details).

Theorem 2. The size of the adaptive dag automaton for a pattern set L is bounded above by $2^{|L|} * |F|$.

Proof. Consider the left-to-right dag automaton for pattern set L. Let M_1 and M_2 be two matching sets labelling two sets S_1 and S_2 in the dag automaton. Assume that M_1 and M_2 have the same rule set. Now, consider the contexts of their relevant items. Since symbols are scanned in the left-to-right order, one of the contexts c_1 must be an

instance of the other context c_2. Hence, there exists a total order among theses prefixes. This argument can be generalised to all the states of the dag automaton that share the same rule set with M_1 and M_2. Consequently, the number of of such prefixes, and hence the number of states, is bounded above by the size of the largest context, which is in turn bounded above $|F|$. Furthermore, as there are $|L|$ distinct patterns, there are at most $2^{|F|}$ different rule sets. This yields the upper bound $2^{|L|}*|F|$ for the size of the adaptive dag automaton. ■

Although the adaptive tree automata that inspect indexes first are smaller (or the same) than the tree left-to-right automaton for any given pattern set, it is not necessarily true that the equivalent adaptive dag automata are smaller than the left-to-right dag automaton. For instance, consider the pattern set $L = \{1{:}f(a,\omega,a,\omega),\ 2{:}f(a,\omega,b,\omega),\ 3{:}f(a,\omega,c,\omega),\ 4{:}f(a,\omega,d,\omega),\ 5{:}f(a,\omega,e,\omega),\ 6{:}f(\omega,b,b,b),\ 7{:}f(\omega,a,\omega,\omega)\}$ where f has an arity of 4 and a, b, c, d and e are constants. Assuming a textual priority rule and using the traversal order, which is described in Section 3, the adaptive dag automaton obtained for L (where most of inspected positions are indexes) is shown in Fig. 5. It has 17 states whereas the left-to-right dag automaton for L, given in Fig. 6, includes 15 states only.

The matching times, however, using the adaptive dag automaton remain better than that using the left-to-right dag automaton as sharing equivalent subautomata does not affect the matching times. In particular, with the adaptive dag automaton of Fig. 5, the matching times for terms that match either of the patterns of all the rules except 6 and 7, are optimal. However, using the left-to-right dag automaton of Fig. 6, every term needs at least three position inspections. Also, from the contrived nature of the example, this situation of the left-to-right dag automata having less states than adaptive ones seems to occur only for rare examples.

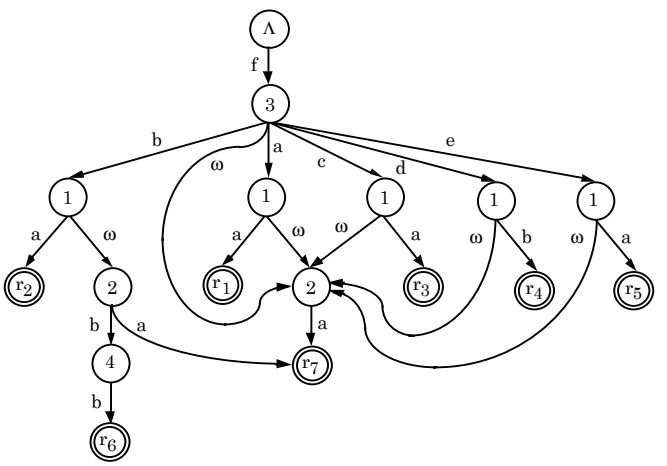


Fig. 5. Adaptive dag automaton for L

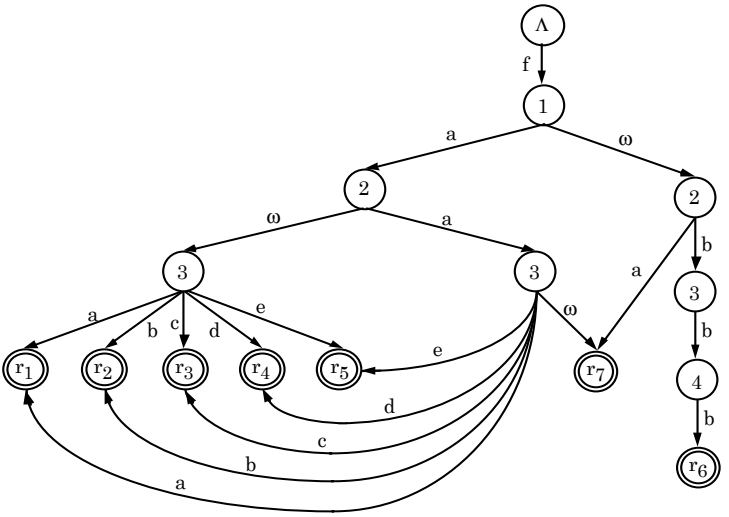


Fig. 6. Left-to-right dag automaton for L

7 Conclusion

First, we described a practical method that compiles a set of prioritised overlapping patterns into an equivalent deterministic adaptive automaton, which does not need backtracking to announce a match. With ambiguous patterns a subject term may be an instance of more than one pattern. To select the pattern to use, a priority rule is usually engaged. The closure operation of the program patterns adds new patterns, which are instances of original ones. Re-examination of symbols while matching terms is completely avoided. The matching automaton can be used to drive the pattern-matching process with any rewriting strategy [11].

In the main body of the paper, we described a method to generate an equivalent minimised dag adaptive matching automaton very efficiently without constructing the tree automaton first. We directly built the dag-based automaton by identifying the states of the tree-based automaton that would generate identical subautomata. By using the dag-based automata we can adaptive pattern-matchers that avoid symbol re-examination without much increase in the space requirements.

Some useful matching set properties were then described for distinguishing different states when building the adaptive dag automaton. A theorem which guarantees equivalence in terms of several simple criteria was then applied to establish improved upper bounds on the size of the dag automaton in terms of just the number of patterns and symbols in the original pattern set.

References

1. Augustsson, A Compiler for Lazy ML, Proceedings ACM Conference on Lisp and Functional Programming, ACM, pp. 218-227, 1984.
2. J. Christian, Flatterms, Discrimination Nets and Fast Term Rewriting, Journal of Automated Reasoning, vol. 10, pp. 95-113, 1993.
3. N. Dershowitz and J.P. Jouannaud, Rewrite Systems, Handbook of Theoretical Computer Science, vol. 2, chap. 6, Elsevier Science Publishers, 1990.
4. A.J. Field and P.G. Harrison, Functional Programming, International Computer Science Series, 1988.
5. Gräf, Left-to-Right Tree Pattern-Matching, Proceedings Conference on Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 488, pp. 323-334, 1991.
6. C.M. Hoffman and M.J. O'Donnell, Pattern-Matching in Trees, Journal of ACM, 29(1):68-95, 1982.
7. P. Hudak and al., Report on the Programming Language Haskell: a Non-Strict, Purely Functional Language, Sigplan Notices, Section S, May 1992.
8. Laville, Comparison of Priority Rules in Pattern Matching and Term Rewriting, Journal of Symbolic Computation, 11:321-347, 1991.
9. N. Nedjah, Pattern-Matching Automata for Efficient Evaluation in Equational Programming, Ph.D. Thesis, University of Manchester-Institute of Science and Technology, Manchester, UK, (Abstract and Contents in the Bulletin of the European Association of Computer Science, vol. 60, November 1997.)
10. N. Nedjah, C.D. Walter and S.E. Eldridge, Optimal Left-to-Right Pattern-Matching Automata, Proceedings of the Sixth International Conference on Algebraic and Logic Programming, Southampton, UK, Lecture Notes in Computer Science, M. Hanus, J. Heering and K. Meinke Editors, Springer-Verlag, vol. 1298, pp. 273-285, 1997.
11. N. Nedjah, C.D. Walter and S.E. Eldridge, Efficient Automata-Driven Pattern-Matching for Equational programs, Software-Practice and Experience, 29(9):793-813, John Wiley, 1999.
12. N. Nedjah and L.M. Mourelle, Improving Time, Space and Termination in Term Rewriting-Based Programming, Proc. International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Budapest, Hungary, Lecture Notes in Computer Science, Springer-Verlag, vol. 2070, pp. 880-890, June 2001.
13. M.J. O'Donnell, Equational Logic as Programming Language, MIT Press, 1985.
14. R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, Adaptive Pattern-Matching, SIAM Journal, 24(6):1207-1234, 1995.
15. D.A. Turner, Miranda: a Non Strict Functional Language with Polymorphic Types, Proceedings of Conference on Lisp and Functional Languages, ACM press, pp. 1-16, 1985.
16. P. Wadler, Efficient Compilation of Pattern-Matching, In "The Implementation of Functional Programming Languages", S. L. Peyton-Jones, Prentice-Hall International, pp. 78-103, 1987.

Adaptive Rule-Driven Devices - General Formulation and Case Study

João José Neto

Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto sn Travessa 3 nro 158
CEP 05508-900 São Paulo SP BRASIL
joao.jose@poli.usp.br

Abstract. A formal device is said to be adaptive whenever its behavior changes dynamically, in a direct response to its input stimuli, without interference of external agents, even its users. In order to achieve this feature, adaptive devices have to be self-modifiable. In other words, any possible changes in the device's behavior must be known at their full extent at any step of its operation in which the changes have to take place. Therefore, adaptive devices must be able to detect all situations causing possible modifications and to adequately react by imposing corresponding changes to the device's behavior. In this work, devices are considered whose behavior is based on the operation of subjacent non-adaptive devices that be fully described by some finite set of rules. An adaptive rule-driven device may be obtained by attaching adaptive actions to the rules of the subjacent formulation, so that whenever a rule is applied, the associated adaptive action is activated, causing the set of rules of the subjacent non-adaptive device to be correspondingly changed. In this paper a new general formulation is proposed that unifies the representation and manipulation of adaptive rule-driven devices and states a common framework for representing and manipulating them. The main feature of this formulation is that it fully preserves the nature of the underlying non-adaptive formalism, so that the adaptive resulting device be easily understood by people familiar to the subjacent device. For illustration purposes, a two-fold case-study is presented, describing adaptive decision tables as adaptive rule-driven devices, and using them for emulating the behavior of a very simple adaptive automaton, which is in turn another adaptive rule-driven device.
Keywords - adaptive devices, rule-driven formalisms, self-modifying machines, adaptive decision tables, adaptive automata.

1 Introduction

Despite many adaptive devices have been created and used in the last decade, notations had not been elaborated in a way that the represented adaptive formalism be as close as possible to the original non-adaptive underlying formulation. Among several other reasons, self-modifying formalisms have not been extensively employed, as a consequence of the complexity of the existing formulations, which make them difficult to use.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 234–250, 2002.
© Springer-Verlag Berlin Heidelberg 2002

In this work we introduce a general proposal for the formulation of adaptive devices based on rule-driven subjacent non-adaptive ones. We expect that the proposed formulation be clear, intuitive and easy to learn, without adding complexity to reading, writing or interpreting the notations. We also expect that the proposal be general enough, in the sense that it be insensitive to the underlying non-adaptive formalism, so that all achieved results remain valid even possibly changing the nature of the subjacent rule-driven system.

2 (Non-adaptive) Rule-Driven Devices

Formal state machines are popular mathematical tools used for describing and modeling actual real life systems. At each stage of their operation, these devices assume some configuration, which usually comprehends the contents of the whole set of information-holding elements and the current status of the device. In conjunction with the input stream yet to be processed, despite how the current configuration was reached, the device's configuration fully determines its further behavior. A formal machine operates by successively moving the device from one configuration to another, in response to stimuli consumed from its input.

Without loss of generality, we may state that such devices start their operation at some initial configuration that follows well-known fixed restrictions. After having processed the full input sequence of stimuli, the device reaches some configuration which may indicate that its whole input stream has been either accepted or rejected. Therefore, we may split the set of all possible configurations for the device into two partitions: one for the accepting configurations and the other for the rejecting ones.

A (*non-adaptive*) *rule-driven device* is any formal machine whose behavior depends exclusively on a finite set of rules which map each possible configuration of the device into a corresponding next configuration. The device is said to be *deterministic* if and only if, for any given initial or intermediate configuration and any input stimulus, its defining set of rules determines one and only one next configuration. The device is *non-deterministic* otherwise. Non-deterministic devices allow more than one valid move at each moment. So, their use requires that all possible next moves be tried as intermediate steps toward some final accepting configuration. Inefficiencies caused by such trial-and-error operation usually turn non-deterministic devices inadequate for sequential implementation. Therefore, in order to achieve efficiency, choosing deterministic equivalent devices is highly recommended. Let us define $ND = (C, NR, S, c_0, A, NA)$ where:

- ND is some *rule-driven device*, described by a set of rules NR.
- C is its set of possible *configurations*, and $c_0 \in C$ is its *initial configuration*.
- S as the (finite) set of all possible *events* that are valid *input stimuli* for ND, with $\varepsilon \in S$.
- $A \subseteq C$ (resp. $F = C - A$) is the subset of its *accepting* (resp. *failing*) *configurations*.
- ε denotes "*empty*", and represents the *null* element of the set it belongs to.

- $w = w_1 w_2 ... w_n$ is a *stream of input stimuli*, where $w_k \in S - \{\varepsilon\}, k = 1, ..., n$ with $n \geq 0$.
- NA is a (finite) set, with $\varepsilon \in NA$, of all possible symbols to be output by ND as side effects to the application of the rules in NR. In practice, *output symbols* in NA may be mapped into procedure calls, so an output generated by applying any rule may be interpreted as a call to its corresponding procedure.
- NR is the set of *rules* defining ND by a relation $NR \subseteq C \times S \times C \times NA$. Rules $r \in NR$ have the form $r = (c_i, s, c_j, z)$, meaning that, in response to any *input stimulus* $s \in S$, r changes the *current configuration* c_i to c_j, consumes s and outputs $z \in NA$ as a side-effect.

A rule $r = (c_i, s, c_j, z)$, with $r \in NR$; $c_i, c_j \in C$; $s \in S$; $z \in NA$, is said to be *compatible* with the current configuration c if and only if $c_i = c$ and s is either empty or equal to the device's current input stimulus. In this case, the application of a single compatible rule moves the device to configuration c_j (denoted by $c_i \Rightarrow^s c_j$) and appends z to its output stream. Note that s, z or both may be empty. Let $c_i \Rightarrow^{\sim} c_m$, $m \geq 0$ denote $c_i \Rightarrow^{\varepsilon} c_1 \Rightarrow^{\varepsilon} c_2 \Rightarrow^{\varepsilon} ... \Rightarrow^{\varepsilon} c_m$, an optional sequence of empty moves. Let $c_i \Rightarrow^{\sim w_k} c_j$, denote $c_i \Rightarrow^{\sim} c_m \Rightarrow^{w_k} c_j$, an optional sequence of empty moves followed by a non-empty one consuming the symbol w_k. An input stream $w = w_1 w_2 \ldots w_n$ is said to be *accepted* by ND when $c_0 \Rightarrow^{\sim w_1} c_1 \Rightarrow^{\sim w_2} ... \Rightarrow^{\sim w_n} c_n \Rightarrow^{\sim} c$ (for short, $c_0 \Rightarrow^w c$, with $c \in A$). Complementarily, w is said to be *rejected* by ND when $c \in F$. The *language* described by ND is the set $L(ND) = \{w \in S^* | c_0 \Rightarrow^w c, c \in A\}$ of all streams $w \in S^*$ that are accepted by ND.

3 Adaptive Devices

Define T as a built-in time counter that starts at 0 and is automatically incremented by 1 when a non-null adaptive action is executed. Each value k assumed by T may subscript the names of time-varying sets. In this case, it selects AD's operation step k.

A device $AD = (ND_0, AM)$ is said to be adaptive whenever, for all operation steps $k \geq 0$, AD follows the behavior of ND_k until the execution of some non-null adaptive action starts its operation step $k + 1$ by changing its set of rules.

At any AD's operation step $k \geq 0$, being ND_k the corresponding subjacent device defined by NR_k, the execution of some non-null adaptive action evolves ND_k to ND_{k+1}. So, AD starts its operation step $k + 1$ by creating the set NR_{k+1} as an edited version of NR_k. Thereafter, AD will follow the behavior of ND_{k+1} until a further non-null adaptive action causes another step to start. This procedure iterates until the input stream is fully processed.

AD starts its operation at c_0, being $AD_0 = (C_0, AR_0, S, c_0, A, NA, BA, AA)$ its initial shape. At step $k \geq 0$, an input stimulus (always) moves AD to a next configuration and then starts its operation step $k + 1$ if and only if a non-adaptive action is executed. So, being AD at step k, with shape $AD_k =$

$(C_k, AR_k, S, c_k, A, NA, BA, AA)$, the execution of a non-null adaptive action leads to $AD_{k+1} = (C_{k+1}, AR_{k+1}, S, c_{k+1}, A, NA, BA, AA)$. In this formulation:

- $AD = (ND_0, AM)$ is some adaptive device, given by an initial subjacent device ND_0 and an adaptive mechanism AM.
- ND_k is AD's subjacent non-adaptive device at some operation step k. ND_0 is AD's initial subjacent device, defined by a set NR_0 of non-adaptive rules. By definition, any non-adaptive rules in any NR_k mirror the corresponding adaptive ones in AR_k.
- C_k is the set of all possible configurations for ND at step k, and $c_k \in C_k$ is its starting configuration at step k. For $k = 0$, we have respectively C_0, the initial set of valid configurations, and $c_0 \in C_0$, the initial configuration for both ND_0 and ND.
- ε ("empty") denotes the absence of any other valid element of the corresponding set.
- S is the (finite, fixed) set of all possible events that are valid input stimuli for AD ($\varepsilon \in S$).
- $A \subseteq C$ (resp. $F = C - A$) is the subset of its *accepting* (resp. *failing*) *configurations*.
- BA and AA are sets of adaptive actions, both containing the null action ($\varepsilon \in BA \cap AA$).
- $w = w_1 w_2 ... w_n$ is a stream of input stimuli, where $w_k \in S - \{\varepsilon\}$, $k = 1, ..., n$.
- NA, with $\varepsilon \in NA$, is a (finite, fixed) set of all possible symbols to be output by AD as side effects to the application of adaptive rules. Just like in non-adaptive devices, the output stream may be interpreted as a corresponding sequence of procedure calls.
- AR_k is a set of adaptive rules, given by a relation $AR_k \subseteq BA \times C \times S \times C \times NA \times AA$. In particular, AR_0 defines the initial behavior of AD. Adaptive actions map the current set of adaptive rules AR_k of AD into a new set AR_{k+1} by adding adaptive rules to AR_k and/or deleting rules from it. Rules $ar \in AR_k$ have the form $ar = (ba, c_i, s, c_j, z, aa)$, meaning that, in response to some input stimulus $s \in S$, ar initially executes the adaptive action $ba \in BA$; the execution of ba is aborted if it eliminates ar from AR_k; otherwise, it applies the subjacent non-adaptive rule $nr = (c_i, s, c_j, z) \in NR_k$, as described before; finally, it executes adaptive action $aa \in AA$.
- Define AR as the set of all possible adaptive rules for AD.
- Define NR as the set of all possible subjacent non-adaptive rules for AD.
- $AM \subseteq BA \times NR \times AA$, defined for a particular adaptive device AD, is an *adaptive mechanism* to be applied at any operation step k to each rule in $NR_k \subseteq NR$. AM must be such that it operates as a function when applied to any sub-domain $NR_k \subseteq NR$. That will determine a single pair of adaptive actions to be attached to each non-adaptive rule. The set $AR_k \subseteq AR$ may be synthesized by collecting all adaptive rules obtained by merging such pairs of adaptive actions to the corresponding non-adaptive rules in NR_k. Equivalently, we may build NR_k by removing all references to adaptive actions from the rules in AR_k.

The algorithm below sketches the overall operation of any rule-driven adaptive device.

1. Set the device at its initial configuration.
2. Set the input stream at its leftmost event.
3. If there are no more events to be processed, then go to step 8
   else feed the device by picking the next event to be processed.
4. Choose the next adaptive rule to be applied:
   Given the current configuration $c_T \in C_T$ and stimulus $s_T \in S$, extracted from the input stream not yet processed, search the set NR_T for compatible rules, i.e., a set of rules that may be applied under the current circumstances, and collect them in a set $CR_T = \{ar \in AR_T | ar = (ba, c_T, s, c, z, aa), s \in \{s_T, \varepsilon\}; c, c_T \in C_T; ba \in BA; aa \in AA; z \in NA\}$. There are three cases to be considered:
   (a) if CR_T is empty, no moves at all are allowed for the device (because AR_T is incompletely specified), then the input stream is rejected by default.
   (b) if $CR_T = \{ar^k\}$ for some $ar^k = (ba^k, c_T, s, c^k, z^k, aa^k) \in AR_T$, then a single rule is available, for deterministic application. By doing so, the device will reach a well-defined next configuration $c_{T+1} = c$.
   (c) if $CR_T = \{ar^k = (ba^k, c_T, s, c^k, z^k, aa^k) | k = 1, 2, ..., m\}$, then all these m rules are equally allowed for being non-deterministically applied to the current configuration, so all of them are applied in parallel to the current configuration, as usual in the operation of non-deterministic devices. Obviously, in non-parallel environments, such parallelism must be exhaustively simulated, e.g. by applying some backtracking strategy.
5. If adaptive action ba^k is the null adaptive action ε in the rule being applied, then proceed to step 6, otherwise, apply the adaptive action ba^k to the current set of rules, yielding a new intermediate configuration for the device AD. Note that, in some cases, even the adaptive rule ar^k being applied may erase itself by executing ba^k. In such an extreme case, the application of ar^p is aborted by returning to step 4.
6. Apply the rule $nr^k = (c_T, s, c^k, z^k)$, just as defined by the underlying non-adaptive device, to the current (intermediate) configuration of AD, yielding another (intermediate) configuration.
7. If the adaptive action aa^k in the adaptive rule ar^k being executed is the null adaptive action ε, proceed to step 3, otherwise apply aa^k to the current set of adaptive rules, finally yielding the next configuration for the device. As in step 5, the execution of the adaptive action may also erase ar^k. In this case, however, no further action is needed.
8. If the current configuration is an accepting one, then accept the input stream, otherwise reject it, then stop.

4 Example

Decision tables are widely accepted tools among information systems programmers and software engineers. Adaptive decision tables constitute an interesting application of the concept of adaptive devices to the field of information systems. Adaptive decision tables may be defined as the class of adaptive devices that use traditional (non-adaptive) decision tables as their underlying formalism.

4.1 (Non-adaptive) Decision Tables

Decision tables may be viewed as tabular devices that encode a set of rules represented by conditions and corresponding actions to be executed when those conditions are matched. In typical decision tables (see Table 1) rules are represented as columns while rows are employed to encode conditions (condition rows) and actions (action rows). In each rule, marked cells corresponding to each condition row refer to relevant conditions to be tested, and indicate whether that condition is to be tested for true (T) or false (F) (non-marked conditions are not to be tested), while marked cells in action rows indicate that the corresponding action is expected to be performed in response to a match in all marked conditions.

Rules in the decision table are encoded as follows:

Condition Rows: all cells of the rule corresponding to conditions to be tested are filled with a boolean value (T or F) corresponding to the particular value to be tested for that condition. A special null mark ($-$) indicates that the associated condition is not to be tested.

Action Rows: cells of the rule corresponding to actions to be performed are filled with T. For all actions not to be executed, these cells are filled with F.

Table 1. Structure of a typical non-adaptive decision table

		1	2	3	4	5	6	7	8	9
Condition rows	c_1	T	T	F	$-$	$-$	F	T	T	$-$
	c_2	$-$	T	T	$-$	$-$	F	$-$	F	T
	...									
	c_n	$-$	$-$	$-$	$-$	T	$-$	F	F	$-$
Action rows	z_1	F	T	T	T	F	F	F	F	T
	z_2	F	F	T	T	F	T	T	F	F
	...									
	z_m	F	F	F	F	T	F	F	T	T

Operation: The operation of such non-adaptive decision tables is quite straightforward:

1. First, the status of the system is checked against the combinations of conditions stated in each of the rules encoded in the table.
2. If no rule matches the current status, then no action is executed at all.
3. If a single rule matches the current status, then we have a deterministic choice, so the matching rule is chosen to be applied.
4. If more than one rule match the current status, then we face a non-deterministic situation. Consequently, all such rules are to be applied in parallel. In practice, parallelism may be simulated, e.g. by some exhaustive backtracking strategy.
5. The selected rule is then applied by executing the set of all actions indicated with a boolean value T in the cells of the rule corresponding to action rows.
6. Once the selected rule has been applied, the decision table gets ready to be used again.

For instance, let us assume that condition c_1 is F and condition c_2 is T. In our decision table, obviously only the rule encoded in column 3 matches such status. Therefore, in this case the decision table will activate actions ra_1 and ra_2 for execution, as we may easily observe by inspecting the action rows specified in rule 3. Note that if rule 1 had been selected instead, no actions would have been called at all, since all action rows in rule 1 are filled with F. It is obvious from this simple example that decision tables are very easy to design and use.

Unfortunately, these classical devices are static, in the sense that their individual rules are all predefined and never change throught the operation of the device. Furthermore, the set of rules defining classical decision tables is not allowed to change, so in classical decision tables there is no dynamic inclusion or exclusion of rules. In order to provide more flexibility to this useful and popular tool, we may use it as the basic underlying non-adaptive formalism for building a far more powerful adaptive device.

4.2 Adaptive Decision Tables

Adaptive decision tables (see Table 2) are easily obtained from conventional (non-adaptive) ones by adding to them further rows encoding the adaptive actions to be performed before ("before-" adaptive action rows) and after the rule is applied ("after-" adaptive action rows).

When adaptive actions are executed, the adaptive table usually has its set of rules modified, therefore correspondingly changing the number of columns in the adaptive decision table. Note that with the chosen layout, however, the number of rows of the adaptive decision tables remains unchanged, since adaptive actions do not modify any of the items encoded in their rows. This property is truly valuable for implementation purposes.

For operating such an adaptive device, the subjacent non-adaptive decision table is first used for determining the rule(s) matching the current situation

Table 2. Structure of an adaptive decision table based on the non-adaptive table in fig. 1 as its subjacent device

		1	2	3	4	5	6	7	8	9
Condition rows	c_1	T	T	F	$-$	$-$	F	T	T	$-$
	c_2	$-$	T	T	$-$	$-$	F	$-$	F	T
	...									
	c_n	$-$	$-$	$-$	$-$	T	$-$	F	F	$-$
Action rows	z_1	F	T	T	T	F	F	F	F	T
	z_2	F	F	T	T	F	T	T	F	F
	...									
	z_m	F	F	F	F	T	F	F	T	T
"before" Adaptive Actions rows	ba_1	T	F	F	F	F	F	F	T	F
	ba_2	F	F	T	F	T	F	F	T	F
	...									
	ba_p	F	T	F	F	F	F	F	F	F
"after" Adaptive Actions rows	aa_1	F	T	T	F	F	F	F	T	F
	aa_2	F	F	F	T	F	T	F	T	F
	...									
	aa_q	F	F	F	F	F	T	F	T	F

of the condition predicates. Then, the selected adaptive rule is performed by executing the indicated "before-" adaptive actions, then applying the subjacent non-adaptive rule, and finally executing the indicated "after-" adaptive actions.

In some cases, when executing its adaptive actions, some adaptive rule being applied may even exclude itself. Whenever the currently used rule happen to be eliminated by its own before- adaptive action, its application is aborted, and the next rule to be applied is elected from the resulting set of rules.

The aspect of an adaptive decision table is shown in the example depicted ahead. The upper half of the table in the figure refers to the corresponding underlying non-adaptive decision table, while its lower half represents the attached adaptive mechanism.

Note that by associating adaptive actions in this way to the usual formulation of decision tables, no substantial changes are introduced for the user, since at first glance adaptive actions might be simply interpreted as additional standard actions to be executed in response to some particular matching of conditions. From a conceptual viewpoint, however, the execution of adaptive actions has significantly deeper implications, since it affects the decision table by allowing changes to be imposed to its own behavior.

4.3 Application

In the table 2 illustrated above, rule 3 is activated when condition c_1 is F and condition c_2 is T, regardless to the other conditions. Under this situation, the application of this adaptive rule operates as follows:

1. Execute adaptive action b_2 (which will probably change the device's set of rules) *before* the subjacent non-adaptive rule is applied.
2. Apply the underlying rule: actions $r_1$and r_2 are performed just as they would be executed in the classical non-adaptive case.
3. Execute adaptive action a_1 (probably changing the decision table again) *after* the application of the underlying non-adaptive rule.

Adaptive Functions Adaptive actions may be defined by means of abstractions called adaptive functions, in a way correspondingly similar to that of function calls and function declarations in a usual programming language. Adaptive functions define generic abstractions while adaptive actions correspond to adaptive function specific calls. Adaptive actions customize the corresponding adaptive function abstraction by assigning arguments to their formal parameters according to each particular needs.

Specifying Adaptive Functions In order to state exactly how each adaptive action is expected to operate, we must provide some further information: the name of the corresponding adaptive function, the set of parameters to be used, the elementary adaptive actions to be applied and the exact way parameters, variables and generators are to be employed.

Thus, the specification of adaptive functions must include the following items:

- *name*: a symbolic name, used for referencing adaptive functions. When calling adaptive actions, the name of the corresponding adaptive function is used to select among available adaptive actions.
- *(formal) parameters*: a set of symbolic names that are used for referencing values passed as arguments to an adaptive function at the time it is called. All instances of the formal symbolic parameters, once replaced with the values associated to their corresponding arguments within the body of the adaptive function, may not be further modified throughout the execution of the adaptive function.
- *variables*: these are symbolic names used for holding values resulting from the application of some rule-searching elementary adaptive function. Variables are filled only once and their values remain unchanged during the execution of the adaptive function.
- *generators*: these elements are symbolic names that refer to new values each time they are used. Once generators are filled with some value, they do not change any more while the adaptive function is active.
- *body*: the body of an adaptive function encodes all editions needed to make the desired changes to the current set of rules of the decision table.

Elementary adaptive actions are editing primitives that allow either testing the rule set or specifying single modifications to the rules of an adaptive decision table. The body of an adaptive decision table consists essentially of a set of elementary adaptive actions.

There are three kinds of elementary adaptive actions that may be combined within the body of an adaptive function in order to specify its operation:

- *rule-searching elementary adaptive actions*: these actions do not modify the set of rules, but allow searching it for rules matching a given pattern.
- *rule-erasing elementary adaptive actions*: these actions remove rules that match a given pattern from the current set of rules.
- *rule-inserting elementary adaptive actions*: these actions allow adding a rule with a specified pattern to the current set of rules.

Encoding Adaptive Functions In order to encode adaptive functions, a format must be chosen for each of the component items listed above. It should be convenient that the format be similar to that of adaptive decision tables. We chose to include the specification of adaptive functions as part of the adaptive decision table itself, since adaptive functions are meaningless outside the environment they act on. The format adopted for encoding adaptive functions within adaptive decision tables will be informally introduced through the following example.

Overall Format for Adaptive Decision Tables Adaptive decision tables as defined here will be drawn as an extension of the notation already discussed for non-adaptive tables. So, the subjacent decision table is represented as usual and the adaptive mechanism is added by inserting the following elements in the rows:

- one heading row containing a tag for specifying the type of each column (H= header of the specification of an adaptive function; $+$, $-$, ? = including, excluding, inspecting elementary adaptive action; S= starting rule; R= normal rule; E= ending rule).
- one extra row for the names of each adaptive function used.
- one extra row for the names of each parameter, variable or generator used by the adaptive functions.
- in the example below, for better legibility of the table, assignments and comparisons referring to variables used by standard (non-adaptive) actions are denoted explicitly and not as function calls.

Similarly, the following additions have been done to the columns of the table:

- one header column for each adaptive function (tag = H) starting the specification of the adaptive function. This header must include a tag B or A in the cell corresponding to the name of the before- or after- adaptive functions, respectively, a tag P in each cell corresponding to a formal parameter, a tag V in each cell corresponding to a variable and a tag G in each cell corresponding to a generator. Each header column is followed by a set of columns related to elementary adaptive actions. This set is finished when a starting rule or another header is found.
- in columns denoting elementary adaptive actions (tags $+$, $-$ or ?) the cells corresponding to conditions are filled with the value the condition is to be tested against; cells corresponding to actions to be executed are marked; cells corresponding to assignments are filled with the value to be assigned. Required adaptive actions are marked, and the cells corresponding to their

parameters are filled with a constant or the name of a variable, a parameter or a generator to be passed as an argument. Homonymous parameters must be avoided between adaptive functions called within the same rule.
- one column for the starting rule (tag = S) of the adaptive automaton, standing for the rule to be applied before any other. In this column, actions are activated in order to initialize all operating conditions for the device. A sequence of normal rules follow this column, ending with the ending rule, which closes the specification of the table.
- columns denoting normal rules (tag = R) specify all rules defining the decision table. Each normal rule is specified by filling condition cells to be tested with constants or names of variables, generators or formal parameters; actions cells and adaptive actions are specified just as described above for elementary adaptive actions.
- a single column, denoting the ending rule (tag = E), serves as a delimiter for the set of current rules in the adaptive decision table, and represents only a logical marker.

Illustrative Example Let us illustrate the encoding of adaptive functions by means of a low-complexity adaptive example. It is shown as a complete adaptive decision table in Fig. 1. In this example, we define two adaptive functions: The first adaptive function is named X and has two formal parameters, $p1$ and $p2$, and uses one generator, $g1$, while the second one, Y, has one formal parameter, $q1$. Note that no variables are used in this example, but if there were variables, new corresponding rows would have been added, since they are denoted and used in the same way parameters and generators are.

Other features that were not used in this example are the calls to before- and after- adaptive actions within adaptive functions. If they were present, additional appropriately tagged columns would have been included for representing them.

The very simple adaptive decision table in Fig.1 illustrates the encoding of the information needed for implementing a simple adaptive device.

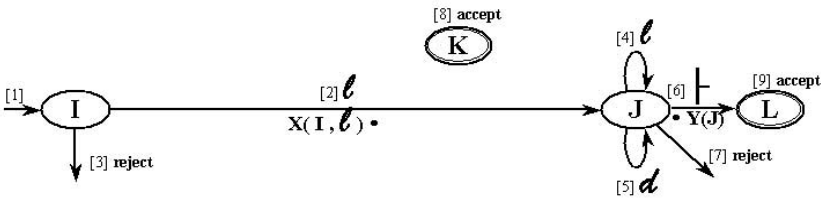
A very brief flash of this adaptive decision table's operation is given in order to illustrate its behavior. In order to shorten the text and help visualizing the evolution of our adaptive device, the automaton represented by the adaptive decision table is represented graphically. New transitions are drawn in heavy lines. Bracketed numbers associated to transitions in the figures refer to the corresponding rule numbers in the adaptive decision table. The initial topology of our automaton is also shown in Fig. 1.

After consuming the first token ℓ from the input stream, one of the automaton's transitions is replaced by five new ones, as shown in Fig. 2.

After consuming one more token, the automaton's shape is changed again, resulting the shape in Fig. 3.

After accepting the whole identifier in the input stream, the resulting automaton is the one depicted in Fig. 4.

In this situation, if a second identifier $\ell d \vdash$ is processed by our adaptive device, the resultant topology of the adaptive automaton will be that shown in Fig. 5.



rule numbers												1	2	3	4	5	6	7	8	9	10
tags		H	+	+	+	+	+	-	H	-	+	S	R	R	R	R	R	R	R	R	E
subjacent table conditions	state=		p1	g1	g1	g1	g1	p1		q1	q1		I	I	J	J	J	J	K	L	
	input=		p2	ℓ	d	⊢		p2		⊢	⊢		ℓ		ℓ	d	⊢				
	state:=		g1	J	J	L		J		L	K	I	J		J	J	L				
subjacent table actions	get		✓									✓	✓		✓	✓					
	accept:=						✘							✘				✘	✓	✓	
adaptive funct. names	X	B		✓	✓			✓					✓								
	Y					✓			A	✓							✓				
parameters	p1	P		g1	g1			p1					I								
	p2	P		ℓ	d			p2					ℓ								
	q1					g1			P	q1							J				
generators	g1	G																			

Fig. 1. Initial adaptive decision table and the corresponding initial topology of the adaptive automaton

Note that after each identifier is fully processed, the path starting at state I and ending at state K will have the shape of a tree, each of whose leaves (transitions pointing to state K) correspond to one different valid identifier found in the input text.

Not all possible options of the model have been explored in this example, but the given illustration is complex enough to be used as a guide for developing other projects with this technique.

This example illustrates the use of adaptive decision tables as an alternative way to implement adaptive automata-based logic. Applications of adaptive automata have been shown in [Jos00]. The target of this decision table is to read an input sequence formed of letters and digits and to collect identifiers formed in the usual way, say, as a non-empty sequence of letters and digits starting with a letter, and ending with the special end-marker ⊢. Whenever a new identifier is found, the adaptive decision table is adequately modified so that the identifier be thereafter registered as an already known one, while already known identifiers are simply accepted by the table without modifying it. So, it operates as a purely syntactic name-collecting device for which all needed available information is permanently encoded in the adaptive decision table.

Further descriptions of the operation of adaptive devices are found in [Jos93].

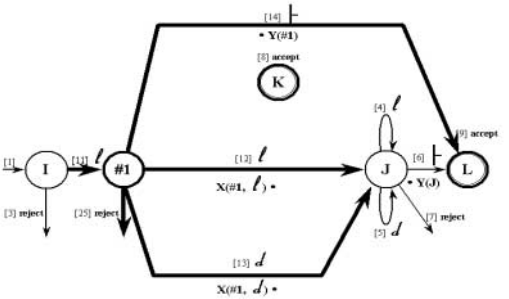


Fig. 2. Adaptive automaton after consuming ℓ

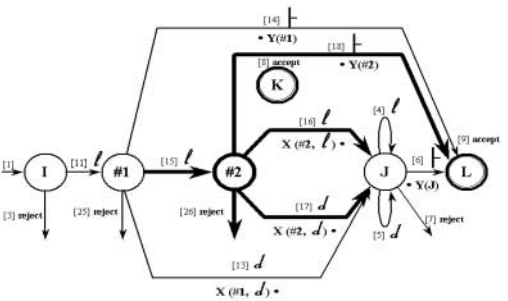
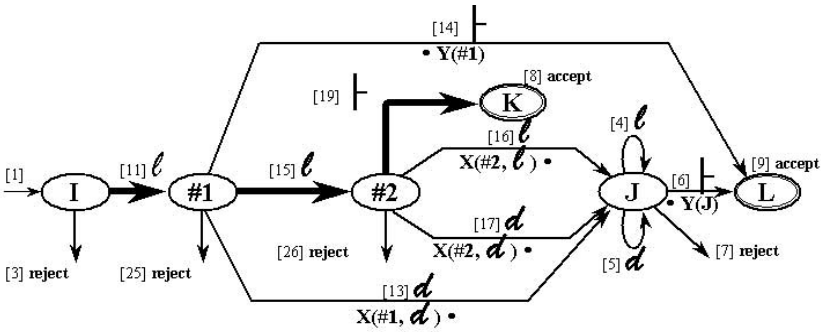


Fig. 3. Adaptive automaton after consuming $\ell\ell$

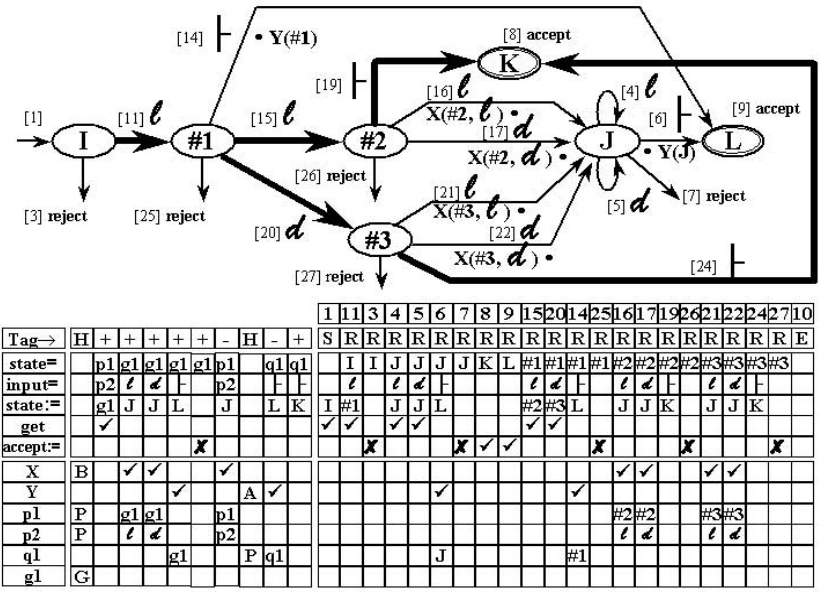
5 Previous Experience with Adaptive Techniques

Adaptive technology concerns to techniques, methods and disciplines referring to actual practical applications of adaptive devices. Historically, adaptive devices emerged from the field of formal languages and automata. Consequently, early applications of such devices have been in the area of rigorous definition of formal and computer languages. In this area, we may list the works by Burshtein [Bur90], Shutt [Shu93], Cabasino [Cab92], Rubinstein [Rub95], Neto [Jos94] and Iwai [Iwa00]. For example, adaptive automata have been proposed as a practical formalism for the representation of languages with context dependencies [Rub95], [Jos94]. On another hand, adaptive grammars were also introduced as generative devices whose operation also allows their use in the rigorous definition of context-dependent languages [Rub95], [Iwa00].

An early meta-system based on adaptive automata has been proposed and implemented which allowed many tests to be carried out in a comfortable form [Per97]. This work was a practical proof that adaptive engines might be useful


[14] ⊢
• Y(#1)
[8] accept
[19] ⊢
K
[16] ℓ
X(#2, ℓ) •
[4] ℓ
[9] accept
[1]
I
[11] ℓ
#1
[15] ℓ
#2
J
[6] ⊢
• Y(J)
L
[17] d
X(#2, d) •
[26] reject
[5] d
[7] reject
[3] reject
[25] reject
[13] d
X(#1, d) •

| | | |



	1	11	3	4	5	6	7	8	9	15	20	14	25	16	17	19	26	21	22	24	27	10
Tag→	S	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	E
state=		I	I	J	J	J	J	K	L	#1	#1	#1	#1	#2	#2	#2	#2	#3	#3	#3	#3	
input=		ℓ		ℓ	d	⊢				ℓ	d	⊢		ℓ	d	⊢		ℓ	d	⊢		
state:=	I	#1		J	J	L				#2	#3	L		J	J	K		J	J	K		
get	✓	✓		✓	✓					✓	✓											
accept:=			✘				✘	✓	✓				✘				✘				✘	
X														✓	✓			✓	✓			
Y						✓						✓										
p1														#2	#2			#3	#3			
p2														ℓ	d			ℓ	d			
q1						J						#1										
g1																						

	H	+	+	+	+	+	-	H	-	+
state=		p1	g1	g1	g1	g1	p1		q1	q1
input=		p2	ℓ	d	⊢		p2		⊢	⊢
state:=		g1	J	J	L		J		L	K
get		✓								
accept:=						✘				
X	B		✓	✓			✓			
Y					✓			A	✓	
p1	P		g1	g1			p1			
p2	P		ℓ	d			p2			
q1					g1			P	q1	
g1	G									

Fig. 5. Adaptive decision table after consuming both $\ell\ell \vdash$ and $\ell d \vdash$ and the corresponding adaptive automaton

education, as a learning mechanism of a computer-aided tutoring system and as a model of the student's progress in such tutoring systems. So, learning is a natural feature shared by all adaptive devices.

In particular, a small experiment with regular languages has been reported that shows the potential of adaptive automata as a good device for constructing grammar inference systems [Jos98]. Another area in which adaptive devices showed their strength is the specification and processing of natural languages. One of the works in this field employed adaptive automata as the main mechanism of an automatic tagging system for texts in Portuguese language. Further works are being currently developed in this direction, with many good intermediate results in the representation of syntactical context-dependent features of natural languages. Simulation and modeling of intelligent systems are also concrete applications of adaptive formalisms, as it was illustrated in the description of the control mechanism of an intelligent autonomous vehicle that collects information from its environment and builds a map for easier navigation. Many other applications for adaptive devices are possible in several fields. They must be extensively explored through the well-succeeded search for simple and efficient alternative ways to perform complex computation tasks.

6 Conclusions

As a result of this work, we have achieved, for adaptive devices, a formulation in which nothing beside the formal adaptive mechanisms is introduced, giving the proposed formulation the character of a simple extension of the underlying non-adaptive device. So, the integrity and even the intuition of the underlying formulation in which the adaptive device is based are preserved, as well as all their properties. Consequently, after getting familiarized with the concept of adaptive devices, users have no extra need of learning further concepts and notations, so they are free for using already designed and tested non-adaptive devices as a basis for easily building adaptive versions. From another viewpoint, with the proposed formulation users may directly identify the underlying non-adaptive device at any moment during the operation of an adaptive device, so simplifying its debugging effort and increasing the comprehension of the adaptive device by its designers. Our proposal has a very clean formulation, allowing the user to be permanently aware of all phenomena concerning the underlying mechanism. No new notations or concepts are introduced, except those involving adaptive features, therefore, our proposal offers a formulation that is indeed intuitive and easy to learn. It is also general to a large extent, since it does not depend on the nature of the underlying non-adaptive formalism chosen. We expect that this simple contribution encourage not only the revisiting and use of existent self-modifying formalisms, but also the formulation of new adaptive devices for solving problems that are hard to solve with usual non-adaptive tools.

Acknowledgements

Our sincere acknowledgement to the anonymous referees for their valuable questions and suggestions for the improvement of this paper.

References

[Alm95] ALMEIDA JUNIOR, J. R. STAD - Uma ferramenta para representação e simulação de sistemas através de statecharts adaptativos. São Paulo 1995, 202p. Doctoral Thesis. Escola Politécnica, Universidade de São Paulo.[In Portuguese] 247

[Bas99] BASSETO, B. A.; JOSÉ NETO, J. A stochastic musical composer based on adaptive algorithms. Anais do XIX Congresso Nacional da Sociedade Brasileira de Computação. SBC-99 PUC-Rio, Vol 3, pp105-13, 19 a 23 de julho de 1999. 247

[Bur90] BURSHTEYN, B. Generation and recognition of formal languages by modifiable grammars. ACM SIGPLAN Notices, v.25, n.12, p.45-53, 1990. 246, 247

[Cab92] CABASINO, S.; PAOLUCCI, P. S.; TODESCO, G. M. Dynamic parsers and evolving grammars. ACM SIGPLAN Notices, v.27, n.11, p.39-48, 1992. 246

[Chr90] Christiansen, H. A survey of adaptable grammars. ACM SIGPLAN Notices, v.25, n.11, p.33-44, 1990. 247

[Iwa00] IWAI, M. K. Um formalismo gramatical adaptativo para linguagens dependentes de contexto. São Paulo 2000, 191p. Doctoral Thesis. Escola Politécnica, Universidade de São Paulo.[In Portuguese] 246

[Jos93] JOSÉ NETO, J. Contribuição à metodologia de construção de compiladores. São Paulo, 1993, 272p. Thesis (Livre-Docência) Escola Politécnica, Universidade de São Paulo.[In Portuguese] 245

[Jos94] JOSÉ NETO, J. Adaptive automata for context-dependent languages. ACM SIGPLAN Notices, v.29, n.9, p.115-24, 1994. 246

[Jos98] JOSÉ NETO, J.; IWAI, M. K. Adaptive automata for syntax learning. XXIV Conferencia Latinoamericana de Informática CLEI'98, Quito - Ecuador, Centro Latinoamericano de Estudios em Informatica, Pontificia Universidad Católica Del Ecuador, tomo 1, pp.135-146. 19 a 23 de Outubro de 1998. 248

[Jos00] JOSÉ NETO, J. Solving Complex Problems Efficiently with Adaptive Automata. CIAA 2000 - Fifth International Conference on Implementation and Application of Automata, July 2000 - London, Ontario, Canada. 245

[Per97] PEREIRA, J. C.D; JOSÉ NETO, J. Um ambiente de desenvolvimento de reconhecedores sintáticos baseados em autômatos adaptativos. II Brazilian Symposium on Programming Languages (SBLP'97), 3-5 September 1997, Institute of Computing, State University of Campinas, Campinas, SP, Brazil, pp.139-50.[In Portuguese] 246

[Rub95] RUBINSTEIN, R. S.; SHUTT. J. N. Self-modifying finite automata: An introduction, Information processing letters, v.56, n.4, 24, p.185-90, 1995. 246

[San97] SANTOS, J. M. N. Um formalismo adaptativo com mecanismos de sincronização para aplicações concorrentes. São Paulo, 1997, 98p. M.Sc. Dissertation Escola Politécnica, Universidade de São Paulo.[In Portuguese] 247

[Shu93] SHUTT, J. N. Recursive adaptable grammar. M. S. Thesis, Computer Science Department, Worcester Polytecnic Institute, Worcester MA, 1993. 246, 247

[Wij75] WIJNGAARDEN, A. V., et al, Revised report on the Algorithmic Language Algol 68, Acta Informatica, v.5, n.1-3, p.1-236, 1975. 247

Typographical Nearest-Neighbor Search in a Finite-State Lexicon and Its Application to Spelling Correction

Agata Savary

LADL, IGM, Université de Marne-la-Vallée
5, bd Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée, France
xsavary@free.fr

Abstract. A method of error-tolerant lookup in a finite-state lexicon is described, as well as its application to automatic spelling correction. We compare our method to the algorithm by K. Oflazer [14]. While Oflazer's algorithm searches for all possible corrections of a misspelled word that are within a given similarity threshold, our approach is to retain only the most similar corrections (nearest neighbours), reducing dynamically the search space in the lexicon, and to reach the first correction as soon as possible.

1 Introduction

K. Oflazer [14] proposed an efficient and elegant algorithm of error-tolerant lookup in a finite-state dictionary, and its application to morphological analysis and spelling correction of simple words. For a given input string that is not contained in the dictionary the algorithm searches for all possible corrections that are within the given distance threshold. We present a similar method in which only those candidates are retained that have the minimal distance from the input word, and the first solution can be obtained rapidly.

2 Related Work

Many aspects of a natural language can be treated through finite-state machines in their classical [16, 7] and extended [8] versions, due to their time and space efficiency obtained by determinisation and minimisation [19, 13, 2].

Automatic spelling correction is one of the oldest applications in the field of natural language processing, and it has a very rich bibliography, a good review of which is presented in [9]. The author divides the existing approaches into three classes: nonword error detection, isolated-word error correction, and context-dependent word correction. Many problems faced by the methods of the first class in the early research (e.g. [12], due to the size of the lexicon and its access time, found a solution in the finite-state model of the lexicon. One of the main remaining problems, the recognition of spelling errors resulting in valid words

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 251–260, 2002.
© Springer-Verlag Berlin Heidelberg 2002

(e.g. *from* → *form*) requires approaches of the third class, based most of the time on a syntactic and/or stochastic analysis of a local context of words supposed to be erroneous (e.g. [17, 5].

In the second type of approach, i.e. isolated error correction, errors are most often of typing origin, of phonetic origin (e.g. [10], or both. This paper addresses only typing errors. They are traditionally interpreted as resulting from one or more *editing operations* on letters: insertions, deletions, replacements and inversions of adjacent letters [3]. Their correction is related to the theoretical problem of approximate string matching [6], in which the distance between two strings is the minimum cost of all sequences of editing operations that transform one string into another. Different sequences of editing operations may be allowed and different cost functions may be assigned to these editing operations. With the distance measure called *edit distance* proposed in [18, 11], editing operations may be assigned arbitrary costs, and they may act on arbitrary positions in the string in arbitrary order (e.g. *ca* can be obtained from *abc* by two operations: deletion of *b*, inversion of *a* and *c*). However, an efficient algorithm for edit distance calculation exists only if $W_I + W_D \leq 2W_S$, where W_S, W_I, W_D are costs assigned to inversion, insertion and deletion operations, respectively.

In [4] this distance measure is modified and renamed to *error distance* by assigning cost 1 to each editing operation and by admitting that errors occur in linear order from left to right so that a later operation may not cancel the effect of an earlier operation. Thus, inversions occur only between letters that are adjacent in the original word and remain adjacent in the erroneous word (e.g. the error distance between *abc* and *ca* is 3). Due to the equal cost of each editing operation, the error distance becomes a *metric*, i.e. a function satisfying four properties: non-negative values, reflexivity, symmetry, and triangular inequality.

The computational solution for the (editing or error) distance calculation, belonging to the class of *dynamic programming* algorithms, is based on a matrix *H[0:n,0:m]*, where n and m are the lengths of the two strings to be compared, and *H[i,j]* contains the distance between the prefixes of lengths i and j of the two strings. The calculation is particularly efficient for the error distance matrix, since the value of the element *H[i+1,j+1]* depends only on the values of the elements *H [i-1,j-1]*, *H[i,j]*, *H[i+1,j]*, and *H[i,j+1]*. Oflazer [14] made the calculation of the error distance matrix even more efficient in that he applied it to the finite-state representation of the lexicon. Thus, when a word is searched for in the lexicon, a part of the matrix is calculated only once for all lexicon words that have the same common prefix.

3 Spelling Correction Problem

The distance measure between two strings admitted in this paper, as well as in Oflazer's one, is the error distance of Du and Chang [4] (although Oflazer still uses the notion of *edit distance*), as described in the previous section. There is no theoretical distance limit between an erroneous word and its right correction. Hence a trade-off is necessary between three factors: the search time efficiency

(in the case of our algorithm and of Oflazer's one it corresponds to the size of the section of the automaton that is to be explored), the length of the resulting correction candidate list (the user may be unwilling to consult a long list), and the chance that the intended word be on that list. Thus, two of the possible spelling correction problem definitions are:

- Finding all valid words which are no more distant from the input word than a given threshold.
- Finding the nearest-neighbours, i.e. the valid words with the minimal distance from the input word (the minimal distance possibly being no bigger than a given threshold).

Note that none of the two approaches guarantees that the right correction will be found. The first approach is more often admitted (e.g. in [4, 14] since the right correction candidate for a misspelled word may not be its nearest neighbour. In our opinion, the second approach is preferable for many applications for three reasons: statistical studies show that words with multiple errors are rare (0.17% till 1.99% of unknown words in a corpus, with [15], users are easily discouraged by long lists of correction candidates, and the search time grows exponentially with the admitted distance threshold. Therefore, the tolerant lookup algorithm we propose finds only the nearest neighbours and concentrates on reaching the first solution (which often is the right one) as soon as possible.

4 Example

The interpretation of a spelling error can be ambiguous. For instance, the erroneous English word **aply* has some one-operation corrections: *apply* (omission of *p*), *paly* (inversion of *p* and *a*), *ply* (insertion of *a*), some 2 two-operation corrections: *ape* (replacement of *e* by *l*, insertion of *y*), *apple* (omission of *p*, replacement of *e* by *y*), *pale* (inversion of *p* and *a*, replacement of *e* by *y*), some three-operation corrections: *apples* (omission of *p* and *s*, replacement of *e* by *y*), *pales* (inversion of *p* and *a*, replacement of *e* by *y*, omission of *s*), etc.

We will show how the nearest neighbours with threshold 2 (in our example these are the one-operation corrections) can be found by an error-tolerant look-up in a deterministic finite-state lexicon. Let us consider a small extract of English lexicon of simple words, containing some possible corrections of **aply* (Fig. 1). The terminal states are represented by double circles. We say that state *w* is reachable from state *v* if there is a transition leading from *v* to *w*. The algorithm follows at first the standard look-up procedure to find the longest correct prefix. It begins in the initial state number 1. Parsing from left to right of *aply* brings us to a non-terminal state 4, where reading the input letter *l* is not possible. Since the automaton is a deterministic one, no backtracking is necessary to be sure that the parsed sequence is not contained in the lexicon. That is where we start the error-tolerant look-up procedure searching for similar words through admission of any of the 4 elementary operations at any of the 5 possible positions in the erroneous word:

a	p	l	y	
1	2	3	4	5

At word position 3, where the standard input blocked, we can make the following suppositions:

- Letter *l* has been wrongly inserted. We omit *l* and try to recognize suffix *-y* starting from the current state 4. That is not possible, so we have to make a second supposition about a possible error. Apart from the wrong insertion of *l*, we may simultaneously have:
  - Wrong insertion of *y*. We try to recognize the empty suffix starting from the current state 4. That is not possible since this state is not a terminal one. No more supposition about a possible error is allowed since we reached the admitted threshold 2.
  - Omission of the correct letter before *y*. We try to recognize suffix *-y* starting from all states reachable from state 4. That is not possible without any further error admission.
  - Replacement of the correct letter by *y*. We consider all transitions leading from state 4 to a final state. There is one such transition: (4,*e*,9). Thus, we get the first two-operation correction candidate *ape* with the error distance 2.
- Letters *l* and *y* have been wrongly inverted. We try to recognize the inverted suffix *-yl* starting from the current state 4 and considering a possible omission at the end of the word (no second error is admitted between *y* and *l* due to the condition that inverted letters must remain adjacent in the target word). That is impossible.
- The correct letter has been omitted at the current position 3. We try to recognize suffix *-ly* starting from any state that is reachable from state 4. In state 9 the recognition of *-ly* is not possible with no more than 1 further error supposition. In state 5 the recognition of *-ly* is possible with no further error supposition, which yields a new 1-operation candidate *apply*. The error distance threshold is reduced to 1. Therefore the 2-operation candidate *apple* is not reached and the previously obtained candidate *ape* is eliminated as it is more distant from the original word than the new candidate.
- The correct letter at position 3 has been replaced by *l*. We try to recognize the suffix *-y* from any state that is reachable from state 4. That is not possible without any further modification. Since the new threshold is 1 this supposition is eliminated.

To continue searching for other candidates we have to backtrack from state 4 to state 2 (and from word position 3 to 2), where 4 possible hypotheses are analysed again: wrong insertion of *p* (suffix *-ly* is unrecognizable from state 2, no candidate is proposed), wrong inversion of *l* and *p* (suffix *-lpy* is unrecognizable), omission of the right letter at position 2 (the only state reachable from state 2 is 4, from which the suffix *-ply* is recognizable yielding the same candidate *apply* as previously obtained), replacement of the right letter at position 2 by *p* (impossible since there is only one transition from 2 to 4).

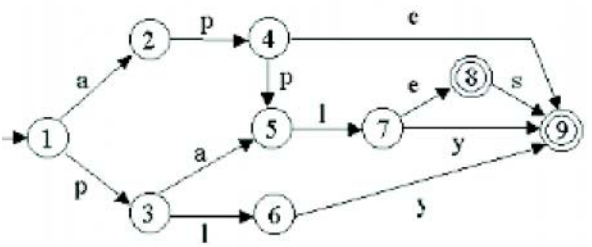


Fig. 1. Extract of a final state lexicon

Finally, backtracking from state 2 to state 1 (and from word position 2 to 1), yields two more one-operation candidates *paly* and *ply*. The two- and three-operation candidates *pale* and *pales* are not reached due to the reduced threshold.

5 Algorithm

An outline of our error-tolerant finite-state lookup algorithm is shown on figure Fig. 5. Let $[l_1 l_2 ... l_n]$ be the word to be looked up, and n its length. Let $wp = 1, ..., n+1$ be the current word position. Let st be the current state, and t the error distance threshold between two suffixes (i.e. the number of elementary operations that we admit in a correct suffix so that it may still be considered a valid correction candidate for a misspelled suffix).

The *tolerant_lookup* function tries to recognize the suffix $[l_{wp}...l_n]$ starting from the current state st and admitting t elementary operations on letters at most. This function returns a pair (ed, S) where S is the set of recognized (exact or modified) suffixes, and ed is the edit distance between the suffix $[l_{wp}...l_n]$ and each of the suffixes in S (all suffixes in S always have the same edit distance from $[l_{wp}...l_n$; if S is empty then $ed = INF$ (a large number, bigger than the maximum edit distance ever possible). The first call to *tolerant_lookup* is done for the entire word $[l_1...l_n]$, the initial state, and the desired edit distance threshold. Then we follow the standard look-up procedure, first without admitting any operation on letters. Thus we can immediately recognize the input word if it belongs to the lexicon, and then quit (the threshold value t becomes 0 in line 9 and lines 13–36 are omitted). If the word doesn't belong to the lexicon the standard look-up ends up with failure in one of the two cases: 1) the input sequence has been read in and the last state is not a terminal one, 2) the input sequence has not been read in completely and no further transition from the current state is possible. If the exact suffix $[l_{wp}...l_n]$ couldn't be recognized, t remains positive (code line 9) and we admit that an error occurred at position wp in the intended word. We try to recognize the input suffix $[l_{wp}...l_n]$ by admitting one of the four elementary operations:

```
1. tolerant_lookup ([l_wp ... l_n],st,t)
2. begin
3. S ← ∅; ed ← INF;
4. if (wp > n)
5.   if terminal(st) then return (0,{ϵ}); endif;
       /*the empty suffix recognized*/
6. endif;

       /*look up the exact suffix, reduce the threshold so as not to admit more
       modifications than in the suffixes already found*/
7.   if (wp ≤ n) and (there is a transition (st,l_wp,st_s))
8.     (ed,S)← tolerant_lookup([l_wp+1 ... l_n],st_s,t);
9.     t = min(t,ed);
       /*concatenate the current letter l_wp
       with all suffixes similar to [l_wp+1 ... l_n]*/
10.    for each (suff ∈ S) do suff ← l_wp ∘ suff; endfor;
11.  endif ;

       /*look up modified suffixes*/
12.  if (t>0)

       /*suppose an insertion at position wp*/
13.    if (wp ≤ n)
14.       (ed_n,S_n)← tolerant_lookup([l_wp+1 ... l_n],st,t-1);
       /*only the suffixes with the smallest edit distance are retained*/
15.       (ed,S) ← add_or_replace(ed,S,ed_n+1,S_n);
16.       t = min(t,ed);
17.    endif;

       /*suppose an inversion of letters at positions wp and wp+1,
       these letters must remain adjacent*/
18.    if ((wp < n) and (l_wp ≠ l_wp+1))
19.       if (∃ ((st,l_wp+1,st_s) and (st_s,l_wp,st_v)))
20.         (ed_n,S_n)← tolerant_lookup([l_wp+2...l_n],st_v,t-1);
21.         for each (suff ∈ S_n) do suff ← [l_wp+1 l_wp] ∘ suff;
22.         endfor;
23.         (ed,S) ← add_or_replace (ed,S,ed_n+1,S_n);
24.         t = min(t,ed);
25.       endif; endif;

26.    for each transition (st,l,st_s)

       /*suppose an omission of a letter at position wp*/
27.       (ed_n,S_n)← tolerant_lookup([l_wp...l_n],st_s,t-1);
28.       for each (suff ∈ S_n) do suff ← l ∘ suff; endfor;
29.       (ed,S) ← add_or_replace (ed,S,ed_n+1,S_n);
30.       t = min(t,ed);

       /*suppose a replacement of the right letter by l_wp if the word not finished*/
31.       if (wp ≤ n)
32.         (ed_n,S_n)← tolerant_lookup([l_wp+1...l_n],st_s,t-1);
33.         for each (suff ∈ S_n) do suff ← l ∘ suff; endfor;
34.         (ed,S) ← add_or_replace (ed,S,ed_n+1,S_n);
35.         t = min(t,ed);
36.       endif; endfor; endif;
37. return(ed,S); end
```

Fig. 2. Error-tolerant lookup algorithm

- Insertion of the letter l_{wp} (if we haven't read the whole word yet; lines 13–17). We omit letter l_{wp} and try to recognize the suffix $[l_{wp+1}...l_n]$ starting from the current state st. We retain only the best solutions (see comment on function *add_or_replace* below).
- Inversion of letters at positions wp and $wp+1$ (if at least two letters are left; lines 18–25). First we try to recognize the inverse infix $[l_{wp+1}l_{wp}]$ starting from the current state st and allowing no modification because we require that inverted letters must remain adjacent. Then we try to recognize the suffix $[l_{wp+2}...l_n]$ starting from the arrival state stv.
- Omission of a letter at position wp (lines 27–30). For each transition leading from the current state st to a state st_s through a label l (line 26), we try to recognize the current suffix $[l_{wp}...l_n]$ starting from the state st_s. Each solution found is concatenated with the transition label l (line 28).
- Replacement of the right letter at position wp through letter l_{wp} (if we haven't read the whole word yet; lines 31–36). For each transition leading from the current state st to a state st_s through label l, we try to recognize the suffix $[l_{wp+1}...l_n]$ starting from the state st_s.

Notice that each time new solutions are found the value of ed and the contents of S are updated by the function *add_or_replace* (lines 15, 23, 29, 34). If new solutions are closer to the original word than the solutions already in S then S is replaced by the set of new solutions, and the value of ed by the new error distance. Otherwise the union of the two sets is done and ed remains unchanged. Thus, only those solutions are retained that have the smallest error distance from the original suffix. Then t gets reduced (lines 9, 16, 24, 30, 35), which limits the range of further searches.

The above algorithm can take as parameter any value of the edit distance threshold, but for languages like English and French, which we tested the program with, the reasonable limit seems to be 2 operations because admitting a bigger edit distance would often result in a great number of irrelevant corrections. Besides the look-up time for a high edit distance threshold would require the exploration of a very big section of the automaton, thus making the search time hardly acceptable for large corpus applications (cf section 6).

6 Complexity and Performance

The exact complexity of our error-tolerant look-up algorithm is difficult to find because it depends not only on the word's length, but also on the size of the dictionary and its precise contents (i.e. the number and length of words that have common subsequences with the input word). Nevertheless, we can make some average case estimation. Let n be the length of the input word, t the error distance threshold, and f_{max} the maximal fan-out of the automaton. Let l_{wp} be the current letter in the input word, s the current state, and f_s the fan-out of s. Depending on what modification is admitted parsing of l_{wp} from state s requires at most:

Table 1. Spelling correction performances

Correction time (**ms**)			
correct sequences	one-error sequences	two-error sequences	sequences with more than 2 errors
7	40	211	233

- 1 transition in case of inversion (the transition that matches l_{wp+1});
- no transition in case of insertion (l_{wp} is omitted, we remain in the current state),
- f_s transitions in case of omission or replacement (all transitions starting from s).

In the worst case, i.e. when the threshold is not reduced during the whole look-up, there are at most $n!/t!(n-t)!$ possible distributions of t modifications over n word positions. For each distribution $(1 + 2 * f_{max})^t$ paths at most must be followed, each path being of length $n+t$ at most. Hence, the worst case complexity is

$$O(n!/t!(n-t)! * (n+t) * 2^t * f_{max}^t).$$

In particular, for t=0 we get $O(n)$, for t=1 $O(n^2 * f_{max})$, for t=2 $O(n^3 * f_{max}^2)$, etc.

We have run the algorithm with threshold 2 on three sets of erroneous strings: sequences belonging to the lexicon, sequences containing one spelling error, and sequences containing two spelling errors or more. The average search time results are presented in the table below. Notice that the correction of 2 errors or more is over 5 times longer than of a single error.

7 Comparison with Oflazer's Algorithm

As we've already mentioned, our algorithm and Oflazer's one admit different definitions of the correction problem (cf section 3).

The main difference though is in the way the calculation of the error (edit) distance is done in the two approaches. In Oflazer's algorithm a matrix H is maintained as described in section 2. Each time a transition is followed in the automaton a new column of the matrix is to be calculated by a function of linear complexity. In our approach the error distance calculation is embedded in the algorithm: each time we admit a modification in the standard lookup procedure the error distance increases. This allows us not to maintain the H-matrix but has also the two major disadvantages:

- It is difficult to adapt the error distance calculation to a particular application or language, e.g. by considering phonetically motivated interchanges of certain letters or groups of letters, as it was done in [1] for Polish.

– A correction candidate may be reached several times with different intermediate error distance values. For example while looking up the word **aply** in the lexicon extract from section 4 with the edit distance threshold 3, the correction candidate *ape* would be first recognized twice as a 3-operation candidate: insertion of l + insertion of y + omission of e, and insertion of l + omission of e + insertion of y. Then the same candidate *ape* would be reached by 2 modifications: insertion of l + replacement of e by y, which would invalidate the two previous solutions. That can make us follow the same path in the automaton several times, which is not time-efficient for bigger values of the edit distance threshold.

For applications in which most errors are of 1 or 2 operations, and in which reaching quickly the first solution is important, our algorithm will often be more efficient due to the fact that we first match the longest correct prefix. Note that in a finite state lexicon the fan-out is very big for the states close to the initial state. Oflazer's algorithm explores most of them at the beginning so it may take a longer time before a solution is found. Our algorithm first skips most of those states (unless the error occurred at the initial position) and follows only the exact path. Since most of misspelled words contain only one error, there is a big chance that the point where the exact path was blocked is the position where the error occurred.

8 Conclusion

We have presented a method of typographical nearest neighbour search in a finite-state lexicon and its comparison to a similar algorithm by Oflazer [14]. Our method is designed for applications where only the least distant corrections are looked for and where the first correction is to be reached as soon as possible. Oflazer's algorithm is simpler and more elegant in the sense that the edit distance calculation is independent from the look-up algorithm.

References

[1] Daciuk, J.: Incremental Construction of Finite-State Automata and Transducers, and Their Use in the Natural Language Processing. Ph.D. Thesis, Politechnika Gdanska, Gdansk (1988) 258

[2] Daciuk, J., Mihov, S., Watson, B., Watson, R.: Incremental Construction of Minimal Acyclic Finite State Automata. Computational Linguistics vol. 26(1). MIT Press, Massachusetts (2000) 3–16 251

[3] Damerau, F. J.: A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM, Vol. 7(3) (1964) 171–176 252

[4] Du, M. W., Chang, S. C.: A model and a fast algorithm for multiple errors spelling correction. Acta Informatica, Vol. 29. Springer Verlag (1992) 281–302 252, 253

[5] Golding, A., Schabes, Y.: Combining Trigram-based and Feature-based Methods for Context-Sensitive Spelling Correction. Proceedings, 34th Annual Meeting of the Association for Computational Linguistics (ACL), Santa Cruz. Association for Computational Linguistics (1996) 71–78 252

[6] Hall, P., Dowling, G.: Approximate String Matching. ACM Computing Surveys, Vol. 12(4). ACM, New York. (1980) 381–402 252
[7] Kaplan, R., Kay, M.: Regular Models of Phonological Rule Systems. Computational Linguistics, Vol. 20(3). Cambridge, Massachusetts, MIT Press (1994) 251
[8] Kornai, A. (ed.): Extended Finite State Models of Language. Cambridge University Press, Cambridge, UK - New York, USA - Melbourne, Australia (1999) 251
[9] Kukich, K.: Techniques for Automatically Correcting Words in Text. ACM Computing Surveys, Vol. 24(4) (1992) 251
[10] Laporte, E., Silberztein, M.: Vérification et correction orthographiques assistées par ordinateur, Actes de la Convention IA 89 (1989) 252
[11] Lowrance, R., Wagner, R. A.: An Extension of the String-to-String Correction Problem. Journal of the ACM, Vol. 22(2) (1975) 177–183 252
[12] McIlroy, M. D.: Development of a Spelling List. IEEE Transactions on Communications, COM-30(1) (1982) 91–99 251
[13] Mohri, M.: Minimization of sequential transducers. Lecture Notes in Computer Science, Vol. 807. Springer Verlag. Berlin. (1994) 251
[14] Oflazer, K.: Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. Computational Linguistics, Vol. 22(1). MIT Press, Cambridge, Massachusetts (1996) 73–89 251, 252, 253, 259
[15] Ren, X., Perrault, F.: The Typology of Unknown Words: An Experimental Study of Two Corpora. Proceedings, 15th International Conference on Computational Linguistics (COLING), Nantes. International Committee on Computational Linguistics (1992) 408–414 253
[16] Roche. E., Schabes, Y. (eds.): Finite-State Language Processing. MIT Press, Cambridge, Massachusetts (1997) 251
[17] Véronis, J.: Morphosyntactic correction in natural language interfaces. Proceedings, 13th International Conference on Computational Linguistics (COLING), Budapest. International Committee on Computational Linguistics (1988) 708–713 252
[18] Wagner, R. A., Fischer, M. J.: The String-to-String Correction Problem. Journal of the ACM, Vol. 21(1) (1974) 168-173 252
[19] Watson, B.: Taxonomies and Toolkits of Regular Language Algorithms. Ph.D. Thesis, Eindhoven University of Technology, the Netherlands (1995) 251

On the Software Design of Cellular Automata Simulators for Ecological Modeling

Yuri Velinov

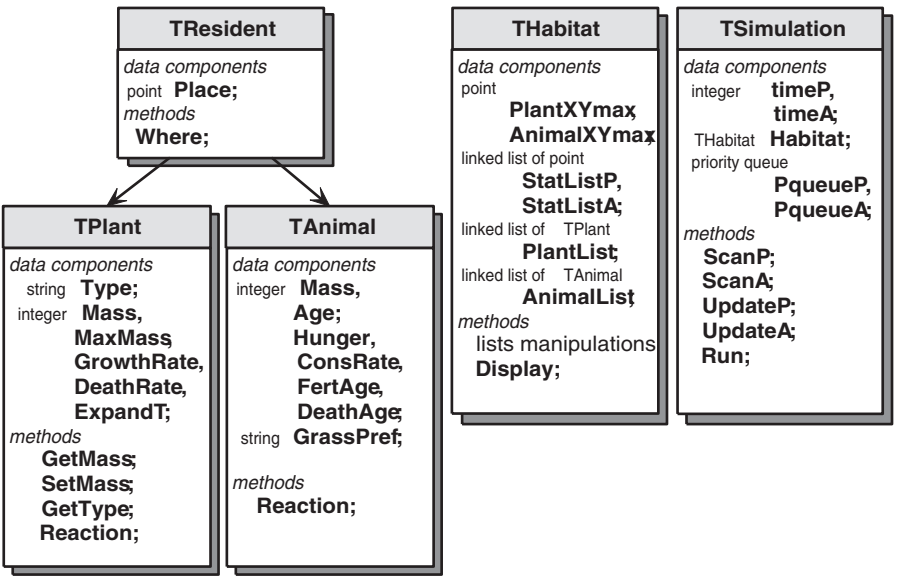
University of Natal, South Africa
yuri@cs.unp.ac.za

The study of arrays of interacting automata situated in the cells of a cellular array (cellular automata) originated in the late forties in the research of John von Neuman. They have found applications in many areas of human activity. In particular, ecological modeling based on cellular automata is suitable for finding adequate solutions in many situations. In this paper we discuss the program implementation of cellular arrays in the framework of the object oriented paradigm using ecological situations for guidelines and inspiration.

The analysis of some typical ecological situations suggests that, in many cases, because of the different nature of the described processes, it is more realistic to consider several overlapping layers of interacting two-dimensional arrays instead of one. Furthermore, the behavior of the environmental objects can frequently be digitalized and modeled by finite automata. On this basis we consider two general modeling approaches - the "active-cells" approach and the "active-residents" approach. The usual active-cells approach associates the activities of a model with its cells and the cells initiate (trigger) all events. As a result cells must be explicitly presented and must be scanned thoroughly one after the other in order to carry out the global dynamic of the model. The active-residents approach associates the activities of a model with the residents of the cells and as a result the events concerning a cell are initiated only if residents are present there. The array of cells is static and can be implicit. The global dynamic of the model can be captured by scanning only the residents. Both approaches can be used for the design of generic or specialized models. The models can be subject to a synchronous dynamic and run following one or several compatible centralized clocks. Alternatively, they can run asynchronously as a sequence of events scheduled by the content of an appropriate priority queue of event-prescriptions. We advocate the active-residents model from the point of view of efficiency.

The flavor of the proposed design approaches can be seen in the following picture, which describes the programming structure of a specialized asynchronous active-residents model for simulation of the grazing behavior of a herd of cows in a grass area. The TResident class of the model specifies only the position of a resident keeping its XY-coordinates (array indexes) in the habitat in the variable Place. It has two descendents - TAnimal and TPlant: TPlant describes the behavior of a patch of grass encapsulated in a cell. It is characterized by the parameters Type, Mass, MaxMass, GrowthRate, DeathRate, and ExpandT. MaxMass gives the maximal amount of biomass, which a cell can hold. If the mass in a cell exceeds the threshold value in ExandT, it can spread on the surrounding cells. TPlant has methods GetMass and SetMass for obtaining in-

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 261–262, 2002.
© Springer-Verlag Berlin Heidelberg 2002



formation about the amount of mass in a cell and adjusting it respectively. The method Reaction returns prescriptions of actions for changes of the parameters of a cell as well as of the parameters of the surrounding cells. Reaction consults a fixed reaction table to determine its outputs. Each prescription contains the coordinates of the place where changes must be enforced. Prescriptions are recorded in a priority queue for processing by the simulation. Alive residents are described in TAnimal in a similar way.

There is no need for an explicit definition of a cell in this model. Cells are presented implicitly in the description of the habitat (THabitat) by fixing the range of their possible positions in CrassXYmax and AnimalXYmax. The components of THabitat include: linked lists StatListA and StatListP of inaccessible points for the cells which do not permit residents; a linked list PlantList of TPlant to represent grass distribution; a linked list AnimalList of TAnimal to represent the description of animals. The method Display demonstrates the situation in the habitat to the user. The dynamic of the model is encapsulated in TSimulation. TSimlation contains time counters timeP and timeA for the plans and animals and two priority queues of event-records. Each event-record contains the moment when the event occurs, the place where the event must happen, and the action to be taken. Each priority queue is filled by the corresponding scan method (ScanA or ScanP), which in turn uses the reaction methods of residents. After the scan at a given moment is completed, the methods UpdateA, UpdateP are invoked to process the corresponding priority queues implementing the events scheduled for earliest time. The method Run of the simulator takes care of the synchronization of processes for plants and animals according to their clocks involving the corresponding Scan and Update methods. The simulation can run in different modes and display the habitat as prescribed by the user.

Random Number Generation with $\oplus$-NFAs

Lynette van Zijl*

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

Abstract. We prove that unary symmetric difference nondeterministic finite automata have the same state cycle as linear feedback shift registers. This leads to the application of these automata for random number generation.

1 Introduction

Nondeterministic finite automata (NFAs) [11] make nondeterministic choices by choosing one path from the union of all possible sets of paths at a given point. Van der Walt [12, 14] introduced selective nondeterministic finite automata ($\star$-NFAs), where the union operator selection can be replaced by an associative and commutative binary operation, $\star$. A $\star$-NFA performs a $\star$ operation on all possible sets of paths, and then makes the nondeterministic choice from the resultant set. The traditional NFA is therefore a special case of a $\star$-NFA with the set operation taken as union. $\star$-NFAs show interesting succinctness properties; these were investigated in detail in [13].

In this paper, we investigate the specific case of $\oplus$-NFAs; that is, $\star$-NFAs where the $\star$ operation is taken as $\oplus$. Here $\oplus$ denotes symmetric difference in the usual set theoretic sense, with $A \oplus B = (A \cup B) \backslash (A \cap B)$.

We formally define $\star$-NFAs and $\oplus$-NFAs in Sect. 2. We then show in Sect. 3 that the behaviour of certain types of $\oplus$-NFAs are similar to that of linear feedback shift registers (LFSRs). This leads to the direct application of $\oplus$-NFAs for random number generation, as discussed in Sect. 4.

2 Definition of $\oplus$-NFAs

Definition 1. *A $\star$-NFA M is a 6-tuple $M = (Q, \Sigma, \delta, q_0, F, \star)$, where Q is the finite non-empty set of states, Σ is the finite non-empty input alphabet, $q_0 \subseteq Q$ is a set of start states and $F \subseteq Q$ is the set of final states. 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 $A \in 2^Q$.

* This research was supported by grants from the University of Stellenbosch.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 263–273, 2002.
© Springer-Verlag Berlin Heidelberg 2002

δ can also be extended to $\delta^* : 2^Q \times \Sigma^* \to 2^Q$ as follows:

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

and

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

for any $a \in \Sigma$, $w \in \Sigma^*$ and $A \in 2^Q$.
To obtain a $\oplus$-NFA, every occurrence of $\star$ is replaced by $\oplus$ in Definition 1 and in the extension of the transition function δ:

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

for any $a \in \Sigma$ and $A \in 2^Q$.[1]

The $\oplus$ operation only retains elements which occur an odd number of times in its operand sets. That is, if $C = A \oplus B$, then C contains only elements which occur an odd number of times in A and B. For example, let $A = \{1, 2, 3\}$ and let $B = \{1, 3, 4\}$. Then $A \oplus B = \{2, 4\}$. In this sense, $\oplus$ is a parity operation, and we define acceptance for $\oplus$-NFAs to reflect this property:

Definition 2. *Let M be a $\oplus$-NFA $M = (Q, \Sigma, \delta, q_0, F, \oplus)$, and let w be a word in Σ^*. Then M accepts w if $q \in q_0$ and $\delta(q, w)$ contains an odd number of final states.*

Other definitions of acceptance were discussed in more detail in [13].

It is easy to see that the definitions of acceptance for the traditional NFA and the $\oplus$-NFA are equivalent if the $\oplus$-NFA has only one final state (that is, $F = \{q\}$ for some $q \in Q$).

A deterministic finite automaton (DFA) is a finite automaton where no non-deterministic choice is allowed; that is, at any given point, the finite automaton can only follow one determined path.

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

Proof. By the well-known subset construction [11], but use (2) to calculate the transition table of the DFA. See [13] for more details.

□

Example 1. Let M be a $\star$-NFA defined by

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

with δ given by (see also Fig. 1)

δ	a
q_1	$\{q_1, q_2\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$.

[1] Hence $\oplus$-NFAs correspond to finite automata with multiplicities in the field F_2 [4].

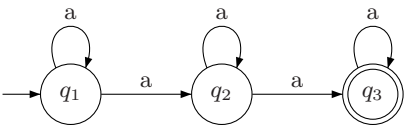


Fig. 1. The ⋆-NFA for Example 1

Choose ⋆ 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.Then 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 ⊕-NFA, the subset construction must be applied using symmetric difference instead of union, and then the transition function δ'' of its equivalent DFA M'' is given by:

δ''	a
$[q_1]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_1, q_3]$
$[q_1, q_3]$	$[q_1, q_2, q_3]$
$[q_1, q_2, q_3]$	$[q_1]$.

□

Definition 3. *A unary ⋆-NFA M is defined as $M = (Q, \Sigma, \delta, \{q_0\}, F, \star)$, where $|\Sigma| = 1$. Without loss of generality, we assume that $\Sigma = \{a\}$ for any unary ⋆-NFA.*

Similarly, a unary DFA M is a DFA with $|\Sigma| = 1$.

In the graphical representation of a unary DFA every node has a single successor (we assume that all entries in the transition table must be specified), 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 (see Fig. 2). 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 (see Fig. 2).

Definition 4. *The state cycle of a unary ⊕-NFA M is the cycle in its equivalent unary DFA M'. The length of the state cycle of M is the length of the cycle of M'.*

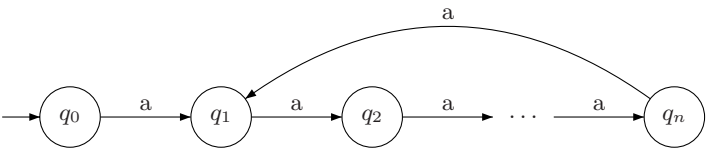


Fig. 2. A cycle of length n in a unary DFA

3 Unary $\oplus$-NFAs and Linear Feedback Shift Registers

In this section we define linear feedback shift registers (LFSRs). We then define an encoding of unary $\oplus$-NFAs which allows us to show a correspondence between the state cycles of certain unary $\oplus$-NFAs and LFSRs [5]. We use this result in the random number generation algorithm in Sect. 4.

3.1 Linear Feedback Shift Registers

An LFSR is a linear autonomous machine over the Galois field GF(2) [3, 5]. In the field GF(2) there are only two elements (0 and 1), and addition and multiplication are defined as

$+$	0	1
0	0	1
1	1	0

and

$\times$	0	1
0	0	0
1	0	1.

Note the resemblance between the $\oplus$ operation and the addition operation in GF(2).

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,

$$\mathbf{Y} = \mathbf{A}\mathbf{y} + \mathbf{B}\mathbf{x} \tag{3}$$

$$\mathbf{z} = \mathbf{C}\mathbf{y} + \mathbf{D}\mathbf{x}. \tag{4}$$

A, B, C and **D** are the characterizing matrices of M.

An *autonomous* linear machine is a linear machine with no input. That is, **B=D=0**, so that the matrix equations (3) and (4) become

$$\mathbf{y}(t) = \mathbf{A}^t\mathbf{y}(0) \tag{5}$$
$$\mathbf{z}(t) = \mathbf{C}\mathbf{y}(t). \tag{6}$$

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 5. *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 6. *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.*

3.2 Encoding a Unary ⊕-NFA

We encode the transition table of a unary ⊕-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

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

Example 2. Let M be the unary ⊕-NFA with the transition table (see also Fig. 3):

δ	a
q_1	$\{q_1, q_2\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_1, q_3\}$.

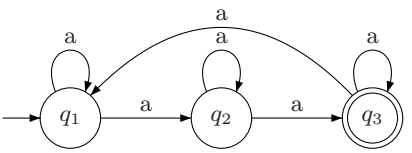


Fig. 3. The unary $\oplus$-NFA M for Example 2

To encode M into the matrix $\mathbf{A}$, encode each row in the transition table into the corresponding column vector. That is, encode $\{q_1, q_2\}$ as $\begin{bmatrix}1\\1\\0\end{bmatrix}$, $\{q_2, q_3\}$ as $\begin{bmatrix}0\\1\\1\end{bmatrix}$, and $\{q_1, q_3\}$ as $\begin{bmatrix}1\\0\\1\end{bmatrix}$ to obtain

$$\mathbf{A} = \begin{bmatrix}1\,0\,1\\1\,1\,0\\0\,1\,1\end{bmatrix}.$$

□

It is easy to show by induction that the first column of $\mathbf{A}^k$ represents the states reached by M after reading the input word a^k.

The interested reader may note the similarities between unary $\oplus$-NFAs and weighted acceptors where the weights are taken in the finite field with two elements [1].

3.3 LFSRs and Unary $\oplus$-NFAs

Theorem 2. *For every n-state LFSR S there exists an $n+1$-state unary $\oplus$-NFA M with the same state cycle as S.*

Proof. Let S be any n-state LFSR with state set $\{q_1, \ldots, q_n\}$. Construct a unary $\oplus$-NFA M as follows. Let the state set Q of M be $Q = \{q_1, \ldots, q_n\}$, and let $\Sigma = \{a\}$. For every $y_i = 1$ in $\mathbf{y}(0)$, let q_i be in the start state set. For $\delta(q_i, a)$, $1 \leq i \leq n$, 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.

□

Example 3. **(LFSRs and $\oplus$-NFAs)**: Let S be an LFSR with characterizing matrix

$$\mathbf{A} = \begin{bmatrix} 0\,0\,0\,1 \\ 1\,0\,0\,1 \\ 0\,1\,0\,0 \\ 0\,0\,1\,0 \end{bmatrix}.$$

It is clear (from the last column of $\mathbf{A}$) that $\mathbf{A}$ is the companion matrix of the polynomial

$$c(X) = X^4 - 0X^3 - 0X^2 - 1X - 1.$$

The interested reader may note that $c(X)$ is a primitive polynomial over GF(2).

The unary ⊕-NFA M with state behaviour equivalent to S is given by

$$M = (\{q_1, q_2, q_3, q_4\}, \{a\}, \delta, \{q_1\}, \{q_4\}, \oplus).$$

The transition table entries are constructed from the matrix $\mathbf{A}$: $q_k \in \delta(q_i, a)$ iff a_{ki} in $\mathbf{A}$ is set to 1, which results in

δ	a
q_1	$\{q_2\}$
q_2	$\{q_3\}$
q_3	$\{q_4\}$
q_4	$\{q_1, q_2\}$.

The DFA M' equivalent to M has $2^4 - 1 = 15$ states and its transition table is given by

δ	a
$[q_1]$	$[q_2]$
$[q_2]$	$[q_3]$
$[q_3]$	$[q_4]$
$[q_4]$	$[q_1, q_2]$
$[q_1, q_2]$	$[q_2, q_3]$
$[q_2, q_3]$	$[q_3, q_4]$
$[q_3, q_4]$	$[q_1, q_2, q_4]$
$[q_1, q_2, q_4]$	$[q_1, q_3]$
$[q_1, q_3]$	$[q_2, q_4]$
$[q_2, q_4]$	$[q_1, q_2, q_3]$
$[q_1, q_2, q_3]$	$[q_2, q_3, q_4]$
$[q_2, q_3, q_4]$	$[q_1, q_2, q_3, q_4]$
$[q_1, q_2, q_3, q_4]$	$[q_1, q_3, q_4]$
$[q_1, q_3, q_4]$	$[q_1, q_4]$
$[q_1, q_4]$	$[q_1]$.

M' can be shown to be minimal.

As an example of the similarity in the state behaviours of S and M, the reader may verify that $\mathbf{y}(5)$ is the encoding of $\delta([q_1, q_2], a)$:

$$\begin{aligned}
\mathbf{y}(5) &= \mathbf{A}^5\mathbf{y}(0) \\
&= \begin{bmatrix} 0&0&0&1 \\ 1&0&0&1 \\ 0&1&0&0 \\ 0&0&1&0 \end{bmatrix}^5 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 0&0&1&1 \\ 1&0&1&0 \\ 1&1&0&1 \\ 0&1&1&0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}
\end{aligned}$$

while $\delta([q_1, q_2], a) = [q_2, q_3]$.

Likewise, $\mathbf{y}(6) = [0\ 0\ 1\ 1]^T$, which corresponds to state $[q_3, q_4]$.

□

Theorem 3. *For any unary $\oplus$-NFA M there is an LFSR with the same state cycle as M.*

Proof. Take any unary $\oplus$-NFA M. Encode its transition table into matrix form to obtain matrix $\mathbf{P}$. Then $\mathbf{P}$ is either non-singular (that is, $\det(\mathbf{P}) \neq 0$) or $\mathbf{P}$ is singular. Consider the first case where $\mathbf{P}$ is non-singular. It is known [3] that either $\mathbf{P}$ is the characterizing matrix of an LFSR, or there exists a matrix $\mathbf{P'}$ similar to $\mathbf{P}$ which has the required form for the characterizing matrix of an LFSR. Recall that an $n \times n$ matrix $\mathbf{G}$ is similar to an $n \times n$ matrix $\mathbf{H}$ iff there is an invertible matrix $\mathbf{J}$ such that $\mathbf{H{=}J}^{-1}\mathbf{GJ}$.

If, on the other hand, $\mathbf{P}$ is singular, then it is known [3] that $\mathbf{P}$ is similar to a matrix $\mathbf{P'}$ in block diagonal form. That is,

$$\mathbf{P'} = \begin{bmatrix} \mathbf{P}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_2 & \dots & \mathbf{0} \\ \dots & \dots & \dots & \dots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{P}_k \end{bmatrix}$$

where the $\mathbf{0}$'s denote whole blocks of zeroes of appropriate size. Here, each $\mathbf{P}_i$ is similar to the characterizing matrix of an LFSR, and $\mathbf{P'}$ represents the union of k unconnected LFSRs. This scenario describes a so-called composite LFSR. In this case the characterizing polynomial is a product of polynomials.

□

4 Random Number Generation with ⊕-NFAs

We showed in the previous section that unary ⊕-NFAs can be characterized by a polynomial over GF(2). This opens a number of well-known applications for unary ⊕-NFAs, such as random number generation, cryptography [10], and others.

We investigated the use of unary ⊕-NFAs as random number generators. Our technique is adapted from LFSR random number generation (see [6, 7, 8] for a description of random number generators based on LFSRs). Random number generation with unary ⊕-NFAs is a six-step process:

1. Take a unary ⊕-NFA M, and encode it into a matrix $\mathbf{A}$ as described in the previous section.
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 $\mathbf{A}^k$ computed above, compute $\mathbf{A}^k\mathbf{y}(0)$. Here $\mathbf{y}(0)$ is the seed for the random sequence, and is formed by encoding the set of 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 of bits.
6. Take each group to represent the binary representation of a whole number. This sequence of numbers forms the pseudo-random sequence.

Example 4. Take M to be the unary ⊕-NFA of Example 3.

Then

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The seed is $\mathbf{y}(0) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, since the start state of M is q_1.

The sequence $\mathbf{A}^k\mathbf{y}(0)$, $k \geq 0$, is given by

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \ldots.$$

Take the bitstream formed by the topmost entries in the column vectors to obtain $1, 0, 0, 0, 1, 0, 0, \ldots$. Then group this sequence of bits into equal-sized groups; a typical choice for the group size is thirty-two, which is the standard word length of the computer.

□

It is trivial to show that the unary $\oplus$-NFAs perform well as random number generators, due to the correspondence between LFSRs and unary $\oplus$-NFAs. As with LFSRs, it can be shown [7, 8] that unary $\oplus$-NFAs with characteristics 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 $\oplus$-NFAs, and taking the symmetric difference of the different bit sequences after the fourth step.

We implemented unary $\oplus$-NFAs as random number generators in the MERLin (**M**odelling **E**nvironment for **R**egular **L**anguages) software environment [2, 15].

We are also investigating the generation of a pseudo-random sequence of $\star$-NFAs in MERLin. In this case we group the bit stream into groups of size mn^2, which is then interpreted as the transition table of an n-state $\star$-NFA with m alphabet symbols. The set of start states and set of final states are generated by two other independent streams, each with group size n. A number of interesting issues arise when pseudo-random sequences of $\star$-NFAs are generated. The first problem here is the question of *disconnected* NFAs. Leslie [9] overcame this problem by manually connecting each generated $\star$-NFA; another approach may be to discard disconnected $\star$-NFAs. Both these approaches change the original pseudo-random bitstream, and therefore compromise the integrity of the random sequence. In MERLin we circumvent the problem by interpreting the results of experiments on random n-state $\star$-NFAs as results on $\star$-NFAs with n or less states (that is, connected and disconnected n-state $\star$-NFAs).

Our method of generating random $\star$-NFAs is based on a mapping from the transition tables of the $\star$-NFAs to numbers – the group of size mn^2 (that is, the size of the transition table as explained above) represents one number in the pseudo-random sequence. However, this does not guarantee in any way that the sequence of generated $\star$-NFAs is random over the domain of the regular languages. We are currently investigating methods to map the pseudo-random $\star$-NFA sequence to an enumeration of the regular languages in order to test the quality of the pseudo-random $\star$-NFA sequence over this domain.

5 Conclusion

We showed that unary $\oplus$-NFAs have state cycles similar to that of LFSRs. This lead to the application of these nondeterministic finite machines as random number generators.

Acknowledgement

Lou Smith implemented the random number generation code in MERLin, and Barney de Villiers, Jeanette Engelmohr, Lesley Raitt and Ryan Wedlake were responsible for system testing and the MERLin User's Manual [2].

References

[1] Culik II, K., Kari, J.: Image Compression using Weighted Finite Automata. Computer and Graphics **17** (1993) 305–313. 268

[2] De Villiers, H. B., Engelmohr, J., Raitt, L., Wedlake, R.: The MERLin User's Manual, Technical Report, Stellenbosch University, March 2001. *http://www.cs.sun.ac.za/projects/techreports* 272

[3] Dornhoff, L. L., Hohn, F. E.: *Applied Modern Algebra.* MacMillan Publishing Co., Inc., New York, 1977. 266, 270

[4] Eilenberg, S.: *Automata, Languages and Machines.* Academic Press, New York, 1974. 264

[5] Golomb, S. W.: *Shift Register Sequences.* Holden-Day, Inc., 1967. 266

[6] L'Ecuyer, P.: Maximally Equidistributed Combined Tausworthe Generators. Mathematics of Computation 65 (1996) 203–213. 271

[7] L'Ecuyer, P.: Tables of Maximally-Equidistributed Combined LFSR Generators. Manuscript. *http://www.iro.umontreal.ca/~lecuyer* 271, 272

[8] L'Ecuyer, P.: Testing Random Number Generators. *Proceedings of the 1992 Winter Simulation Conference*, IEEE Press, Dec. 1992, 305–313. 271, 272

[9] Leslie, T. K. S.: *Efficient Approaches to Subset Construction.* MSc Thesis, University of Waterloo, Waterloo, Canada, 1994. 272

[10] Salomaa, A.: *Public-Key Cryptography.* Springer-Verlag, Berlin, 1990. 271

[11] Sipser, M.: *Introduction to the Theory of Computation.* PWS Publishing Company, 1997. 263, 264

[12] Van der Walt, A., Van Zijl, L.: ⋆-Realizations of Deterministic Finite Automata. British Colloquium for Theoretical Computer Science **11**, Swansea, Wallis, April 1995. 263

[13] Van Zijl, L.: *Generalized Nondeterminism and the Succinct Representation of Regular Languages.* Ph.D. dissertation, Stellenbosch University, March 1997. 263, 264

[14] Van Zijl, L., Van der Walt, A. P. J.: Some Automata-Theoretic Properties of ∩-NFAs. South African Computer Journal **24** (1999) 163–167. 263

[15] Van Zijl, L., Harper, J.-P., Olivier, F.: The MERLin Environment Applied to ⋆-NFAs. *Proceedings of the CIAA2000*, London, Ontario, Canada, July 2000. To appear in Lecture Notes in Computer Science **2088** (2001). 272

[16] Wolfram, S.: *Cellular Automata and Complexity.* Addison-Wesley, 1994.

Supernondeterministic Finite Automata

Lynette van Zijl*

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

Abstract. We show that a simple generalization of the transition tables of nondeterministic finite automata leads to a hierarchy of succinct nondeterministic descriptions for finite automata. We show that the hierarchy corresponds to deterministic finite automata on level 0 and nondeterministic finite automata on level 1 by default, and prove that the hierarchy corresponds to alternating (boolean) finite automata on level 2. We show that there exists an n-state level 3 finite automaton M such that its equivalent minimal deterministic finite automaton M' has more that 2^{2^n} states.

1 Introduction

A deterministic finite automaton (DFA) allows no choice in its movements; that is, in a given state and with a given input alphabet symbol, the DFA can only make one predetermined move. On the other hand, a nondeterministic finite automaton (NFA) allows nondeterministic choice in its movements, so that in a given state and with a given input alphabet symbol, the NFA can choose from a set of possible moves. This behaviour is reflected in the transition tables of these automata, with the transition function δ of a DFA defined as a function from the state set Q of the DFA to Q. For an NFA, the transition function δ takes the state set Q of the NFA to subsets of Q (that is, the power set $\mathcal{P}(Q)$ of Q). Alternation [2] is a generalization of nondeterminism, and in an alternating automaton (AFA), the transition function δ maps the state set Q of the AFA to boolean functions on Q. It is well-known that NFAs are exponentially more succinct than DFAs [6], and AFAs are double-exponentially more succinct than DFAs [2, 4]. The question arises whether it would be possible to find a description mechanism which would be triple-exponentially more succinct than DFAs, or even better. The purpose of this paper is to describe and investigate such a generalization of nondeterminism which we call *supernondeterminism*.

We show that supernondeterministic finite automata (sNFAs) provide a uniform mechanism to describe any level of exponential succinctness over DFAs. We define level k supernondeterministic finite automata (k-sNFA) in Sect. 2. In Sect. 3 we discuss equivalences between 0-sNFAs and DFAs, between 1-sNFAs and NFAs, and between 2-sNFAs and alternating (boolean) automata. We also show in Sect. 4.2 the existence of an n-state 3-sNFA with an equivalent minimal DFA with more than 2^{2^n} states.

* This research was supported by grants from the University of Stellenbosch.

B.W. Watson and D. Wood (Eds.): CIAA 2001, LNCS 2494, pp. 274–288, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Definition of k-sNFAs

To find a hierarchy of succinct description mechanisms for finite automata, we generalize the principle employed in the transition function of an NFA. We let the zero-th level of the hierarchy correspond to finite automata where the transition function takes states into states; for the first level, where the transition function takes states into sets of states; for the second level, where the transition function takes states into sets of sets of states; and for the k-th level, where the transition function takes states into the k-th powerset of the state set.

Assume that k is some fixed non-negative integer, and Q is some finite non-empty universe.

Let $\mathcal{P}^j(A)$ indicate j power set applications on a set A:

Definition 1. *Let $A \subseteq Q$ be any finite set, and let $\mathcal{P}(A)$ be the power set of A. Then $\mathcal{P}^j(A)$ is defined recursively as*

$$\begin{aligned} \mathcal{P}^1(A) &= \mathcal{P}(A) \\ \mathcal{P}^j(A) &= \mathcal{P}(\mathcal{P}^{j-1}(A)), \quad j \geq 2. \end{aligned}$$

□

Definition 2. *A k-sNFA is a 6-tuple $M = (Q, \Sigma, \delta, q_0, F, k)$ where Q is the finite non-empty state set, Σ is the finite non-empty input alphabet, δ is the transition function $\delta : Q \times \Sigma \to \mathcal{P}^k(Q)$, $q_0 \in Q$ is the start state and $F \subseteq Q$ is the final state set.* □

In order to extend the transition function δ to $\delta : \mathcal{P}^k(Q) \times \Sigma \to \mathcal{P}^k(Q)$, one needs to capture the difference between transitions such as (for $a \in \Sigma$)

$$\delta(\{[q_1], [q_2]\}, a)$$

and

$$\delta(\{[q_1, q_2]\}, a).$$

To this end, we identify the lowest-level elements in $\mathcal{P}^k(Q)$ (which we call atomic sets), and group these together in a cross product operation to preserve the differences as above.

Definition 3. *For any set $A \in \mathcal{P}^j(Q)$, $1 \leq j \leq k$, define the* atomic set of A *to mean a set A_{j-1}, with $A_{j-1} \in \mathcal{P}(Q)$, such that there exists a sequence of sets $A_{j-1}, A_{j-2}, \ldots, A_1$ with*

$$A_{j-1} \in A_{j-2} \in \ \ldots \ \in A_1 \in A,$$

and each $A_s \in \mathcal{P}^{j-s}(Q)$, $1 \leq s \leq j-1$.

Similarly, let atomic-i set *denote a set A_{i-1} with $A_{i-1} \in \mathcal{P}^{j-(i-1)}(Q)$ such that*

$$A_{i-1} \in A_{i-2} \in \ \ldots \ \in A_1 \in A,$$

and each $A_s \in \mathcal{P}^{j-s}(Q)$, $1 \leq s \leq i-1$. □

Example 1. Let $Q = \{1, 2, 3\}$ and $A = \{\{\{1, 2\}, \{1\}\}, \{\{3\}\}\}$. Then $A \in \mathcal{P}^3(Q)$. The atomic sets of A are the three sets $\{1, 2\}$, $\{1\}$ and $\{3\}$, while the set $\{\{1, 2\}, \{1\}\}$ is an example of an atomic-2 set. □

The following definition allows for the retrieval of all the atomic sets of a set A:

Definition 4. *Let A be a set such that $A \in \mathcal{P}^j(Q)$, with $1 \leq j \leq k$. Then*

$$\odot(A) = \{A\}, \qquad \text{for } A \in \mathcal{P}(Q),$$
$$\odot(A) = \{B \in \mathcal{P}(Q) \mid B \text{ is an atomic set of } A\} \qquad \text{for } A \in \mathcal{P}^j(Q), \quad 2 \leq j \leq k.$$

□

Note that $\odot(A)$ is always an element of $\mathcal{P}^2(Q)$, irrespective of the value of j for which $A \in \mathcal{P}^j(Q)$.

We now define the cross product operation on sets.

Definition 5. *Let A and B be sets such that $A \in \mathcal{P}^i(Q)$ and $B \in \mathcal{P}^j(Q)$, with $1 \leq i, j \leq k$. Suppose that $\odot(A) = \{A_1, \ldots, A_{m_1}\}$ and $\odot(B) = \{B_1, \ldots, B_{m_2}\}$. Then*

$$A \otimes B = \{A_1 \cup B_1, A_2 \cup B_1, \ldots, A_{m_1} \cup B_{m_2}\}$$

$A \otimes B$ is called the cross-star of A and B. □

We also define a unary cross-star operation:

Definition 6. *Let A be a set such that $A \in \mathcal{P}^j(Q)$, with $1 \leq j \leq k$. Then*

$$\otimes A = \odot(A).$$

□

Note that, if $A \in \mathcal{P}^2(Q)$, then $\otimes(A) = \odot(A) = A$.

Example 2. (a) Let $A = \{1, 2\}$ and $B = \{3\}$. Then $\odot(A) = \{\{1, 2\}\}$ and $\odot(B) = \{\{3\}\}$. Therefore,

$$A \otimes B = \{\{1, 2\} \cup \{3\}\} = \{\{1, 2, 3\}\}.$$

(b) Let $A = \{\{1, 2\}, \{1\}\}$ and $B = \{\{2\}, \{3\}\}$. Then $\odot(A) = \{\underbrace{\{1, 2\}}_{A_1}, \underbrace{\{1\}}_{A_2}\}$ and $\odot(B) = \{\underbrace{\{2\}}_{B_1}, \underbrace{\{3\}}_{B_2}\}$. Therefore,

$$\begin{aligned} A \otimes B &= \otimes(\underbrace{\{\{1, 2\}, \{1\}\}}_{A}, \underbrace{\{\{2\}, \{3\}\}}_{B}) \\ &= \{\bigcup(\underbrace{\{1, 2\}}_{A_1}, \underbrace{\{2\}}_{B_1}), \bigcup(\underbrace{\{1, 2\}}_{A_1}, \underbrace{\{3\}}_{B_2}), \bigcup(\underbrace{\{1\}}_{A_2}, \underbrace{\{2\}}_{B_1}), \bigcup(\underbrace{\{1\}}_{A_2}, \underbrace{\{3\}}_{B_2})\} \\ &= \{\{1, 2\}, \{1, 2, 3\}, \{1, 3\}\}. \end{aligned}$$

□

Theorem 1. *The cross-star operation $\otimes$ is commutative and associative.*

Proof. The result follows directly from the commutativity and associativity of the union operation. □

The transition function δ can now be extended to $\delta : \mathcal{P}^k(Q) \times \Sigma \to \mathcal{P}^k(Q)$ as follows: Assume that $A \in \mathcal{P}^j(Q)$ with $2 \leq j \leq k$, and that $A = \{A_1, \ldots, A_m\}$ with $A_i \in \mathcal{P}^{j-1}(Q)$, $1 \leq i \leq m$. Then, for any $a \in \Sigma$,

$$\begin{aligned}
\delta(A, a) &= \bigcup(\otimes_{q \in A_1} \delta(q, a), \ldots, \otimes_{q \in A_m} \delta(q, a)), \quad \text{for } A \in \mathcal{P}^2(Q) \\
\delta(A, a) &= \delta(\{A_1, \ldots, A_m\}, a) \\
&= \{\delta(A_1, a), \ldots, \delta(A_m, a)\}, \qquad \text{for } A \in \mathcal{P}^j(Q),\ 2 < j \leq k.
\end{aligned}$$

In other words, to calculate the transition function on a set $A \in \mathcal{P}^j(Q)$, we 'bubble' down to the level of the atomic sets of A, and calculate the cross-star on the atomic sets.

The transition function δ can also be extended to $\delta : \mathcal{P}^k(Q) \times \Sigma^* \to \mathcal{P}^k(Q)$. Assume that $A \in \mathcal{P}^j(Q)$, with $1 \leq j \leq k$. Then, for any $a \in \Sigma$ and $w \in \Sigma^*$,

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

and

$$\delta(A, aw) = \delta(\delta(A, a), w).$$

Example 3. Suppose $Q = \{q_1, q_2, q_3\}$. Let M be a 3-sNFA such that

$$M = (Q, \{a\}, \delta, q_1, \{q_3\}, 3)$$

with δ given by

δ	a
q_1	$\{\{\{q_1\}, \{q_1, q_2\}\}, \{\{q_3\}\}\}$
q_2	$\{\{\{q_3\}\}\}$
q_3	$\{\{\{q_1, q_3\}\}\}$.

Then

$$\begin{aligned}
&\delta(\{\{\{q_1\}, \{q_1, q_2\}\}, \{\{q_3\}\}\}, a) \\
=&\{\ \delta(\{\{\{q_1\}, \{q_1, q_2\}\}, a),\ \delta(\{\{\{q_3\}\}, a)\} \\
=&\{\ \bigcup\ (\ \otimes(\delta(q_1, a)), \otimes(\delta(q_1, a), \delta(q_2, a))),\ \cup(\otimes\delta(q_3, a))\} \\
=&\{\ \bigcup\ (\ \otimes(\{\{\{q_1\}, \{q_1, q_2\}\}, \{\{q_3\}\}\}), \otimes(\{\{\{q_1\}, \{q_1, q_2\}\}, \{\{q_3\}\}\}, \{\{\{q_3\}\}\})), \\
&\quad \cup(\otimes(\{\{\{q_1, q_3\}\}\}))\} \\
=&\{\ \bigcup\ (\ \{\{q_1\}, \{q_1, q_2\}, \{q_3\}\},\ (\{\{q_1\} \cup \{q_3\}, \{q_1, q_2\} \cup \{q_3\}, \{q_3\} \cup \{q_3\}\})), \\
&\quad \{\{q_1, q_3\}\}\} \\
=&\{\ \bigcup\ (\ \{\{q_1\}, \{q_1, q_2\}, \{q_3\}\},\ \{\{q_1, q_3\}, \{q_1, q_2, q_3\}, \{q_3\}\}), \\
&\quad \{\{q_1, q_3\}\}\} \\
=&\{\{\{q_1\}, \{q_1, q_2\}, \{q_3\}, \{q_1, q_3\}, \{q_1, q_2, q_3\}\},\ \{\{q_1, q_3\}\}\}.
\end{aligned}$$

□

The example above highlights the main characteristic of the cross-star operation: It generates many different combinations of the atomic sets that occur in the transition table of the k-sNFA. Also, the combined effect of the definitions of $\delta(A, a)$ and the cross-star operation ensures that atomic-i sets are joined together into atomic-i sets in the desired manner. Without the cross-star operation, the k-sNFAs cannot improve on the $O(2^n)$ upper bound of NFAs, since one would then be able to construct only combinations of r items from n, which results in only

$$\binom{n}{0} + \binom{n}{1} + \ldots + \binom{n}{r} + \ldots + \binom{n}{n} = 2^n$$

possibilities.

Acceptance in a k-sNFA is defined as:

Definition 7. *Let M be a k-sNFA $M = (Q, \Sigma, \delta, q_0, F, k)$, and let w be a word in Σ^*. Then M accepts w if $\delta(q_0, w)$ contains at least one final state $f \in F$ in every one of its atomic sets. That is, if $\delta(q_0, w) = A \in \mathcal{P}^k(Q)$, then for every atomic set A^i_{k-1} of A there must exist an $f^i \in F$ such that $f^i \in A^i_{k-1} \in A^i_{k-2} \in \ \ldots \ \in A$.* □

In the rest of this article we refer to definition 7 as the *traditional definition of acceptance*.

Theorem 2. *Let $\mathcal{L}(M)$ be a language accepted by a k-sNFA M. Then there exists a DFA M' that accepts $\mathcal{L}(M)$.*

Proof. Similar to the traditional case for NFAs [3]; the reader is referred to [7] for a detailed proof. □

Example 4. Take the 2-sNFA $M = (Q, \{a\}, \delta, q_1, \{q_3\}, 2)$, with δ defined as

δ	a
q_1	$\{\{q_1, q_2\}, \{q_3\}\}$
q_2	$\{\{q_3\}\}$
q_3	$\{\{q_2, q_3\}\}$.

Then the DFA equivalent to M is given by

δ'	a
$[[q_1]]$	$[[q_1, q_2], [q_3]]$
$[[q_1, q_2], [q_3]]$	$[[q_1, q_2, q_3], [q_3], [q_2, q_3]]$
$[[q_1, q_2, q_3], [q_3], [q_2, q_3]]$	$[[q_1, q_2, q_3], [q_2, q_3]]$
$[[q_1, q_2, q_3], [q_2, q_3]]$	$[[q_1, q_2, q_3], [q_2, q_3]]$.

We show one of the steps in more detail:

$$\begin{aligned}
\delta(\{\{q_1, q_2\}, \{q_3\}\}, a) &= \bigcup \ (\otimes_{q \in \{q_1, q_2\}} \delta(q, a) \ , \ \otimes_{q \in \{q_3\}} \delta(q, a)) \\
&= \bigcup \ (\otimes(\{\{q_1, q_2\}, \{q_3\}\} \ , \ \{\{q_3\}\}) \ , \ \otimes(\{\{q_2, q_3\}\})) \\
&= \bigcup \ (\{\{q_1, q_2, q_3\}, \{q_3\}\}, \ \{\{q_2, q_3\}\}) \\
&= \{\{q_1, q_2, q_3\}, \{q_3\}, \{q_2, q_3\}\}.
\end{aligned}$$

□

3 Equivalences

A DFA is by default equivalent to a 0-sNFA, and an NFA to a 1-sNFA. We now prove that a 2-sNFA is equivalent to an alternating (boolean) automaton.

We briefly recall the properties of a boolean automaton. For more detail, see [1].

A boolean automaton is a 5-tuple $M = (Q, \Sigma, \delta, f^0, F)$ where Q is the finite set of states, Σ is the finite non-empty input alphabet, δ is the transition function $\delta : Q \times \Sigma \to \{0,1\}^{\{0,1\}^Q}$ (that is, δ takes $Q \times \Sigma$ into the set of all boolean functions on Q), f^0 is the start function and $F \subseteq Q$ is the set of final states. A word $w \in \Sigma^*$ is accepted if the final state set F satisfies $\delta(f^0, w)$. To translate a boolean automaton into an equivalent DFA $M' = (Q', \Sigma, \delta', q'_0, F')$, take all $2^{2^{|Q|}}$ boolean functions on Q and label these appropriately to form the state set Q'. The transition function δ' is defined as

$$\delta'(f, a) = f(\delta(q_1, a), \ldots, \delta(q_n, a)).$$

That is, $\delta'(f, a)$ is the function f applied to all n transitions on the alphabet symbol a.

A DFA function-labelled state $[f]$ is accepting if the final state set F satisfies the function f (that is, substitute into f a *true* for each state in F and a *false* for each state not in F – if f evaluates to *true*, $[f]$ is a final state of the DFA).

In order to prove equivalence between a 2-sNFA and a boolean automaton, we first define the *boolean encoding* of a set in $\mathcal{P}^2(Q)$, and then prove a lemma concerning the cross-star operation.

Definition 8. *Let A, B be sets such that $A \in \mathcal{P}(Q)$ and $B \in \mathcal{P}^2(Q)$, where $A = \{a_1, a_2, \ldots, a_s\}$ and $B = \{B_1, B_2, \ldots, B_m\}$, with $B_i = \{b_{i,1}, \ldots, b_{i,m_i}\}$ for $1 \leq i \leq m$. Then the* boolean encoding *$e(A)$ of A is defined as*

$$e(A) = a_1 \vee \ \ldots \ \vee a_s$$

and the boolean encoding $e(B)$ of B is defined as

$$\begin{aligned} e(B) &= e(B_1) \wedge e(B_2) \wedge \ \ldots \ \wedge e(B_m) \\ &= (b_{1,1} \vee \ldots \vee b_{1,m_1}) \wedge (b_{2,1} \vee \ldots \vee b_{2,m_2}) \wedge \ \ldots \ (b_{m,1} \vee \ldots \vee b_{m,m_m}) \\ &= \wedge_{i=1}^{m} (b_{i,1} \vee \ \ldots \ \vee b_{i,m_i}) \\ &= \wedge_{i=1}^{m} \vee_{j=1}^{m_i} b_{i,j}. \end{aligned}$$

□

Lemma 1. *Let B be a set such that $B \in \mathcal{P}^2(Q)$, with $B = \{B_1, \ldots, B_m\}$ and $B_i = \{b_{i,1}, \ldots, b_{i,m_i}\}$ for $1 \leq i \leq m$. Suppose there is a set $C \subseteq Q$ which satisfies $e(B)$. Then there must be at least one $c \in C$ such that $c \in B_i$ for every $1 \leq i \leq m$.*

Proof. We know that $e(B) = \wedge_{i=1}^{m} \vee_{j=1}^{m_i} b_{i,j}$. Therefore, for any set C which satisfies $e(B)$, there must be at least one element $c \in C$ which satisfies every conjunct of $e(B)$. The result holds. □

Lemma 2. *(a) Let A, B be sets such that $A, B \in \mathcal{P}(Q)$, with $A = \{a_1, \ldots, a_{m_1}\}$ and $B = \{b_1, \ldots, b_{m_2}\}$. Then $e(A \cup B) = e(A) \vee e(B)$.*

(b) Let A, B be sets such that $A, B \in \mathcal{P}^2(Q)$, where $A = \{A_1, A_2, \ldots, A_{m_1}\}$ and where $B = \{B_1, B_2, \ldots, B_{m_2}\}$. Then $e(A \cup B) = e(A) \wedge e(B)$.

Proof. Trivial.

Corollary 1. *Let B_i be sets such that $B_i \in \mathcal{P}^2(Q)$. Then*

$$e(\bigcup_{i=1}^{m} B_i) = \wedge_{i=1}^{m} e(B_i).$$

Lemma 3. *Let A, B be sets such that $A, B \in \mathcal{P}^2(Q)$, with $A = \{A_1, A_2, \ldots, A_{m_1}\}$ and $B = \{B_1, B_2, \ldots, B_{m_2}\}$. Then*

$$e(\otimes(A, B)) = e(A) \vee e(B).$$

Proof. By Lemma 2, and the distributive law. □

Corollary 2. *Let B_i be sets such that $B_i \in \mathcal{P}^2(Q)$. Then*

$$e((\otimes)_{i=1}^{m} B_i) = \vee_{i=1}^{m} e(B_i).$$

Lemma 4. *Let M be any n-state 2-sNFA $M = (Q, \Sigma, \delta, q_0, F, 2)$ and let M' be a boolean automaton with $M' = (Q', \Sigma, \delta', f^0, F')$. Suppose that $Q' = Q$, $f^0 = q_0$, $F' = F$ and δ' is constructed from δ such that for every entry $\delta(q, a) = \{A_1, \ldots, A_m\}$, it holds that $\delta'(q, a) = \wedge_{i=1}^{m}(\vee_s(q_s \in A_i))$. Then for any set $B \in \mathcal{P}^2(Q)$*

$$e(\delta(B, a)) = \delta'(e(B), a).$$

Proof. From the construction of δ' it is clear that $\delta'(q, a) = e(\delta(q, a))$ for every $q \in Q$.

Suppose that $B = \{A_1, A_2, \ldots, A_m\}$, where each set $A_i = \{a_{i1}, a_{i2}, \ldots, a_{im_i}\}$ for $1 \leq i \leq m$. Then

$$\begin{aligned} e(B) &= e(\{A_1, \ldots, A_m\}) \\ &= \wedge_{i=1}^{m} e(A_i) \\ &= \wedge_{i=1}^{m} \ \vee_{a \in A_i} a. \end{aligned}$$

In the case of the boolean automaton M', by definition,

$$\delta'(f, a) = f(\delta'(q_1, a), \ldots, \delta'(q_n, a)).$$

Therefore,

$$\begin{aligned} \delta'(e(B), a) &= e(B)(\delta'(q_1, a), \ldots, \delta'(q_n, a)) \\ &= \wedge_{i=1}^{m} e(A_i)(\delta'(q_1, a), \ldots, \delta'(q_n, a)) \\ &= \wedge_{i=1}^{m} e(A_i)(e(\delta(q_1, a)), \ldots, e(\delta(q_n, a)) \\ &= \wedge_{i=1}^{m} \ \vee_{q \in A_i} e(\delta(q, a)). \end{aligned}$$

But for the 2-sNFA M we know that

$$\begin{aligned}\wedge_{i=1}^{m} \ \vee_{q\in A_i} \ e(\delta(q,a)) &= \wedge_{i=1}^{m} e((\otimes)_{q\in A_i}\delta(q,a)) && \text{(by Corollary 2)}\\ &= e(\textstyle\bigcup_{i=1}^{m}(\otimes)_{q\in A_i}\delta(q,a))) && \text{(by Corollary 1)}\\ &= e(\delta(B,a)) && \text{(by the definition of } \delta).\end{aligned}$$

Hence, $e(\delta(B,a)) = \delta'(e(B),a)$. □

Theorem 3. *(a) Any 2-sNFA can be translated to a boolean automaton, with the same number of states, which accepts the same language.*

(b) Any boolean automaton can be translated to a 2-sNFA with at most twice the number of states, which accepts the same language.

Proof. (a) Consider any n-state 2-sNFA $M = (Q, \Sigma, \delta, q_0, F, 2)$. Let

$$M' = (Q', \Sigma, \delta', f^0, F')$$

be the boolean automaton constructed as follows: Set $Q' = Q$, $f^0 = q_0$, and $F' = F$. Construct δ' from δ so that, for every entry $\delta(q,a) = \{A_1, \ldots, A_m\}$, we have $\delta'(q,a) = \wedge_{i=1}^{m}(\vee_s(q_s \in A_i))$. That is, $\delta'(q,a) = e(\delta(q,a))$.

We now claim that M and M' accept the same language. By induction on the length of a word $w \in \Sigma^*$, it is easy to see that

$$e(\delta(q_0,w)) = \delta'(e(q_0),w)$$

for any $w \in \Sigma^*$.

It follows that M and M' accept the same language: Let $w \in \Sigma^*$ such that $\delta'(f^0,w) = f'$. Suppose w is accepted by M'. Then f' is satisfied by the set $F = F'$. But if f' is satisfied by F, then $e(\delta(q_0,w))$ is satisfied by F (since $e(\delta(q_0,w)) = \delta'(e(q_0),w) = f'$). Therefore, by Lemma 1, there must be at least one element $q_f \in F$ such that q_f is an element of every atomic set in $\delta(q_0,w)$. That is, w is accepted by M.

Conversely, suppose that $w \in \Sigma^*$ is accepted by M. Then there must be at least one element $q_f \in F$ which occurs in every atomic set in $\delta(q_0,w)$. By the definition of the encoding of a set in $\mathcal{P}^2(Q)$, if q_f occurs in every atomic set in $\delta(q_0,w)$, then q_f occurs in every disjunct in the conjunctive normal form $e(\delta(q_0,w))$. Therefore any set A such that $q_f \in A$ satisfies $e(\delta(q_0,w))$. In particular, since $q_f \in F'$, it follows that F' satisfies $e(\delta(q_0,w)) = \delta'(f^0,w)$.

The result holds.

(b) Take any boolean automaton $M = (Q, \Sigma, \delta, f^0, F)$. Construct a 2-sNFA as follows: For every $q \in Q$, let q and $\overline{q}$ be states in Q', and let $q_0 \in Q'$ be a start state (q_0 is not in Q). Let $F' = F$. To construct δ', encode any boolean function on Q as a set in $2^{2^{Q'}}$ such that $q_i \wedge q_j$ is encoded as $\{\{q_i\},\{q_j\}\}$ and $q_i \vee q_j$ is encoded as $\{\{q_i,q_j\}\}$. Let $\delta'(q_0,\epsilon)$ be the encoded set associated with f^0, and $\delta'(q_0,a)$ be undefined for every $a \in \Sigma$. For every other $q \in Q$, let $\delta'(q,a)$ be the encoded set associated with $\delta(q,a)$ for all $a \in \Sigma$. For all $q \in Q$ such that $\overline{q} \in Q'$, let $\delta'(\overline{q},a)$ be the encoded set associated with $\overline{\delta(q,a)}$.

The argument to show that M and M' accept the same language now holds as in part (a) of this proof. □

Note that, instead of the translation rule $\delta'(q,a) = \wedge_{i=1}^{m}(\vee_s(q_s \in A_i))$, we can use the rule $\delta'(q,a) = \vee_{i=1}^{m}(\wedge_s(q_s \in A_i))$. In that case, choose $F = \overline{F'}$. Then the 2-sNFA accepts if all and only the final states occur in at least one atomic set. This is the well-known notion of *universal acceptance*.

Example 5. We take an example from [5] to illustrate the simulation of a boolean automaton by a 2-sNFA.

Consider the DFA A_m defined by

$$A_m = (\{0,1,\ldots,m-1\},\{a,b,c\},\delta,0,\{0\})$$

with δ given by

$$\begin{aligned}
\delta_m(i,a) &= (i+1) \bmod m \\
\delta_m(i,b) &= \begin{cases} 1 & ,i=0 \\ 0 & ,i=1 \\ i & ,i=2,3,\ldots,m-1 \end{cases} \\
\delta_m(i,c) &= \begin{cases} m-1 & ,i=0 \\ i & ,i=1,2,\ldots,m-1. \end{cases}
\end{aligned}$$

The minimal DFA A_m^R of the reverse of A_m has 2^m states, and the boolean automaton B_m accepting the same language as A_m^R has $\log m$ states. We choose $m = 8$ and (after some manipulation) find B_8:

$$B_8 = (\{1,2,3\},\{a,b,c\},\mu,\overline{1} \wedge \overline{2} \wedge \overline{3},\emptyset)$$

with μ given by (abbreviate $x \wedge y$ by xy)

μ	a	b	c
1	$\overline{1}23 \vee 1\overline{2}3 \vee 12\overline{3}$	1	$1 \vee \overline{123}$
2	$\overline{12}3 \vee \overline{1}2\overline{3} \vee 1\overline{2}3 \vee 12\overline{3}$	2	$2 \vee \overline{123}$
3	$\overline{3}$	$\overline{123} \vee \overline{1}23 \vee 1\overline{2}3 \vee 123$	$3 \vee \overline{123}$.

The minimal DFA equivalent to B_8 has 256 states.

Using the translation algorithm given in Theorem 3 above, we can write down the 2-sNFA M equivalent to B_8. Note that we demonstrate the alternative translation rule $\delta'(q,a) = \vee_{i=1}^{m}(\wedge_s(q_s \in A_i))$ instead of $\delta'(q,a) = \wedge_{i=1}^{m}(\vee_s(q_s \in A_i))$.

$$M = (\{1,2,3,\overline{1},\overline{2},\overline{3}\},\{a,b,c\},\mu',\{\overline{1},\overline{2},\overline{3}\},\{\overline{1},\overline{2},\overline{3}\},2)$$

with μ' defined by

μ'	a	b	c
1	$\{\{\overline{1},2,3\},\{1,\overline{2},\overline{3}\},\{1,\overline{2},3\},\{1,2,\overline{3}\}\}$	$\{\{1\}\}$	$\{\{1\},\{\overline{1},\overline{2},\overline{3}\}\}$
2	$\{\{\overline{1},\overline{2},3\},\{\overline{1},2,\overline{3}\},\{1,\overline{2},3\},\{1,2,\overline{3}\}\}$	$\{\{2\}\}$	$\{\{2\},\{\overline{1},\overline{2},\overline{3}\}\}$
3	$\{\{\overline{3}\}\}$	$\{\{\overline{1},\overline{2},\overline{3}\},\{\overline{1},2,3\},\{1,\overline{2},3\},\{1,2,3\}\}$	$\{\{3\},\{\overline{1},\overline{2},\overline{3}\}\}$
$\overline{1}$	$\{\{\overline{1},\overline{2},\overline{3}\},\{1,2,3\},\{\overline{1},2,\overline{3}\},\{\overline{1},\overline{2},3\}\}$	$\{\{\overline{1}\}\}$	$\{\{\overline{1},2\},\{\overline{1},3\}\}$
$\overline{2}$	$\{\{\overline{1},\overline{2},\overline{3}\},\{1,2,3\},\{\overline{1},2,3\},\{1,\overline{2},\overline{3}\}\}$	$\{\{\overline{2}\}\}$	$\{\{1,\overline{2}\},\{\overline{2},3\}\}$
$\overline{3}$	$\{\{3\}\}$	$\{\{1,\overline{2},\overline{3}\},\{\overline{1},2,\overline{3}\},\{\overline{1},\overline{2},3\},\{1,2,\overline{3}\}\}$	$\{\{1,\overline{3}\},\{2,\overline{3}\}\}$

The example above uses multiple start states; it could be replaced by a dummy state 0, with $\mu'(0,\sigma) = \{\overline{1},\overline{2},\overline{3}\}$ for all $\sigma \in \Sigma$.

We refrain from listing the DFA. However, testing some short input strings illustrates the equivalence: For B_8 and input string ϵ,

$$\begin{aligned}\mu(f^0,\epsilon) &= f^0(\mu(1,\epsilon),\mu(2,\epsilon),\mu(3,\epsilon))\\ &= \overline{1}\wedge\overline{2}\wedge\overline{3}\end{aligned}$$

which satisfies the final state set $F = \emptyset$ and hence B_8 accepts the empty set.

On the same input string ϵ, since M has the start state $\{\{\overline{1},\overline{2},\overline{3}\}\}$, it accepts ϵ.

On the input string a, B_8 behaves as follows:

$$\begin{aligned}\mu(f^0,a) &= f^0\big(\mu(1,a),\mu(2,a),\mu(3,a)\big)\\ &= \overline{1}\wedge\overline{2}\wedge\overline{3}\big(\mu(1,a),\mu(2,a),\mu(3,a)\big)\\ &= (\overline{123}\vee 123\vee\overline{1}23\vee\overline{12}3)\wedge(\overline{123}\vee 123\vee\overline{1}23\vee 1\overline{23})\wedge(3).\end{aligned}$$

Substituting a 1 for every $x \in \{\overline{1},\overline{2},\overline{3}\}$ and a 0 for every $x \in \{1,2,3\}$, we see that $\mu(f^0,a) = 1\wedge 1\wedge 0 = 0$ and hence the string a is not accepted by B_8.

For M on input string a, it is obvious that $\otimes(\mu'(\overline{1},a),\ \mu'(\overline{2},a),\ \mu'(\overline{3},a))$ must necessarily have a 3 in every atomic set (since $\mu'(\overline{3},a) = 3$) and hence cannot contain an atomic set $\{\overline{1},\overline{2},\overline{3}\}$. It follows that M does not accept the string a.

□

4 Existence of a Succinct 3-sNFA

4.1 k-sNFAs for $k > 2$, with Traditional Acceptance

With the traditional definition of acceptance, we can show that any k-sNFA, for all $k > 2$, behaves just like a 2-sNFA. We show in Sect. 4.2 that *generalized acceptance* is needed in order to find a k-sNFA which is $O(\underbrace{2^{2^{\cdot^{\cdot^{2^n}}}}}_{k\text{ times}})$ succinct.

To proof that any k-sNFA behaves like a 2-sNFA, we need some preliminary definitions and lemmas.

We first extend Definition 8 to sets in $\mathcal{P}^j(Q)$.

Definition 9. *Consider* $A \in \mathcal{P}^j(Q)$. *Let* $A = \{a_1, \ldots, a_m\}$ *for* $j = 1$, *and* $A = \{B_1, \ldots, B_m\}$ *for* $j > 1$, *where* $B_i \in \mathcal{P}^{j-1}(Q)$ *for* $1 \leq i \leq m$. *Then the boolean encoding* $e(A)$ *of* A *is defined as*

$$e(A) = \begin{cases} \vee_i (a_i \in A) & \text{for } A \in \mathcal{P}^1(Q) \\ \wedge_{i=1}^m \; e(B_i) & \text{for } A \in \mathcal{P}^j(Q)\,, \; j \geq 2. \end{cases}$$

□

A consequence of this definition is that $e(A)$ is always in conjunctive normal form.

Conversely, given a boolean formula with variables from Q, we may associate with it a set $D \in \mathcal{P}^2(Q)$:

Definition 10. *Let* f *be any boolean formula over* Q *such that* f *is in conjunctive normal form, and* f *contains no negated variables; say*

$$\begin{aligned} f &= \wedge_{i=1}^s \;\; \vee_{j=1}^t q_{s,t} \\ &= (q_{1,1} \vee \ldots \vee q_{1,t}) \wedge \;\; \ldots \;\; \wedge (q_{s,1} \vee \ldots \vee q_{s,t}). \end{aligned}$$

Then the set D *associated with* f *is defined as* $D = d(f)$ *where*

$$D = \{\{q_{1,1}, \ldots, q_{1,t}\}, \ldots, \{q_{s,1}, \ldots, q_{s,t}\}\}.$$

□

Lemma 5. *Let* $A \in \mathcal{P}^j(Q)$, $j > 2$, *and suppose that the atomic sets of* A *are* $A_1, \ldots, A_m$. *Then* $e(d(e(A))) = e(A)$.

Proof. From the definitions of $d()$ and $e()$, we make the simple observation that for any set $A \in \mathcal{P}^j(Q)$, $d(e(A))$ is just the set of all the atomic sets of A. The result follows. □

Lemma 6. *Let* A, B *be sets such that* $A, B \in \mathcal{P}^j(Q)$, *with* $A = \{A_1, A_2, \ldots, A_{m_1}\}$ *and* $B = \{B_1, B_2, \ldots, B_{m_2}\}$. *Then* $e(A \cup B) = e(A) \wedge e(B)$.

Proof. A trivial extension of Lemma 3(b).

Corollary 3. *Let* B_i *be sets such that* $B_i \in \mathcal{P}^j(Q)$. *Then*

$$e(\bigcup_{i=1}^m B_i) = \wedge_{i=1}^m e(B_i).$$

Lemma 7. *Let* M *be any* n*-state* k*-sNFA* M, *with* $k > 2$, *and* M' *an* n*-state* 2*-sNFA such that* $Q' = Q$, $q'_0 = q_0$, *and* $F' = F$. *Also, let* $\delta'(q, a) = d(e(\delta(q, a)))$ *for every* $q \in Q'$ *and* $a \in \Sigma$. *Suppose now that* B *is any set in* $\mathcal{P}^k(Q)$ *with atomic sets* $A_1, \ldots, A_m$. *Then* $d(e(\delta(B, a)))$ *is exactly* $\delta'(d(e(B)), a)$.

Proof. We know that for any set $B \in \mathcal{P}^k(Q)$, $d(e(B))$ is just the set of all the atomic sets of B. Also, since the set of atomic sets of $\delta(q,a)$ is the same as the set of atomic sets of $d(e(\delta(q,a)))$, it follows that

$$\otimes\delta(q,a) = \otimes d(e(\delta(q,a))).$$

Also, from the recursive nature of the definition of $\delta(B,a)$ for $B \in \mathcal{P}^k(Q)$, using Corollary 3, it follows directly that

$$e(\delta(B,a)) = \wedge_{i=1}^{m} \otimes\delta(A_i,a)).$$

Therefore

$$\begin{aligned}
\delta'(d(e(B)),a)) &= \delta'(\{A_1,\ldots,A_m\},a) \\
&= \bigcup_{i=1}^{m} \otimes_{q\in A_i} \delta'(q,a) \\
&= \bigcup_{i=1}^{m} \otimes_{q\in A_i} d(e(\delta(q,a))) \\
&= \bigcup_{i=1}^{m} \otimes_{q\in A_i} \delta(q,a) \\
&= d(e(\bigcup_{i=1}^{m} \otimes_{q\in A_i} \delta(q,a))) \\
&= d(e(\delta(B,a))).
\end{aligned}$$

The lemma holds. □

Theorem 4. *Any n-state k-sNFA, for all $k > 2$, can be reduced to a 2-sNFA with at most n states, under the traditional definition of acceptance.*

Proof. Take any n-state k-sNFA M, with $k > 2$. We wish to construct an n-state 2-sNFA M' equivalent to M. Let $Q' = Q$, $q_0' = q_0$, and $F' = F$. To construct the transition function δ', we let $\delta'(q,a) = d(e(\delta(q,a)))$ for every $q \in Q'$ and $a \in \Sigma$.

It remains to show that M and M' accept the same language.

We use induction on the length of a string $w \in \Sigma^*$ to prove that

$$d(e(\delta(q_0,w))) = \delta'(q_0,w)$$

for any $w \in \Sigma^*$.

The result holds trivially for $|w| = 0$, since $q_0 = q_0'$, and hence $d(e(q_0)) = q_0 = q_0'$. Suppose now that for any string w of length t, it holds that

$$d(e(\delta(q_0,w))) = \delta'(q_0,w).$$

Then consider the string $wa \in \Sigma^*$:

$$\begin{aligned}
d(e(\delta(q_0,wa))) &= d(e(\delta(\delta(q_0,w),a))) & \\
&= d(e(\delta(A,a))) & \text{(where } A = \delta(q_0,w) \in \mathcal{P}^k(Q)\text{)} \\
&= \delta'(d(e(A)),a) & \text{(by Lemma 7)} \\
&= \delta'(d(e(\delta(q_0,w))),a) & \\
&= \delta'(\delta'(q_0,w),a) & \text{(by the induction hypothesis)} \\
&= \delta'(q_0,wa). &
\end{aligned}$$

Now, M accepts a word w if there is at least one final state in every atomic set in $\delta(q_0,w)$. In that case, $e(\delta(q_0,w))$ must contain a final state in every conjunct, and hence $d(e(\delta(q_0,w)))$ contains a final state in every atomic set. It follows that M' then also accepts w. The converse argument follows similarly, and the theorem therefore holds. □

4.2 k-sNFAs for $k > 2$, with Generalized Acceptance

We defined acceptance for k-sNFAs in an existential manner: A word $w \in \Sigma^*$ is accepted if every atomic set in $\delta(q_0, w)$ contains at least one final state. Under this definition, we were able to demonstrate in Theorem 4 that any k-sNFA for $k > 2$ is equivalent to a 2-sNFA, and hence no more than $O(2^{2^n})$ succinct. We now define a new form of acceptance called generalized acceptance, and show that under generalized acceptance this restriction does not hold.

Definition 11. *Let M be a k-sNFA $M = (Q, \Sigma, \delta, q_0, F, k)$, where F is chosen to be $F \subseteq \mathcal{P}^k(Q)$, and let w be a word in Σ^*. Then M accepts w if $\delta(q_0, w) \in F$.* □

The notion of generalized acceptance basically allows the final state set to be chosen from the states of the DFA, instead of the states of the k-sNFA.

Theorem 5. *There is a 2-state 3-sNFA which has an equivalent minimal DFA, under generalized acceptance, with more than 2^{2^2} states.*

Proof. Define a 3-sNFA M such that

$$M = (\{q_1, q_2\}, \{a, b, c\}, \delta, q_1, F, 3)$$

with δ defined by

δ	a	b	c
q_1	$\{\{\{q_2\}\}\}$	$\{\{\{q_2\}\}\}$	$\{\{\{q_2\}\}\}$
q_2	$\{\{\{q_1, q_2\}\}\}$	$\{\{\{q_1\}\}, \{\{q_2\}\}\}$	$\{\{\{q_1\}, \{q_2\}\}\}$.

Choose the final state set (under generalized acceptance) as

$$\begin{aligned} F = \{\ & \{\{\{q_1, q_2\}\}\}, \\ & \{\{\{q_1\}, \{q_2\}, \{q_1, q_2\}\}\}, \\ & \{\{\{q_2\}\}, \{\{q_1, q_2\}\}, \{\{q_2\}, \{q_1, q_2\}\}\}, \\ & \{\{\{q_1\}, \{q_2\}\}, \{\{q_2\}, \{q_1, q_2\}\}\}, \\ & \{\{\{q_1\}, \{q_2\}\}, \{\{q_1\}, \{q_2\}, \{q_1, q_2\}\}\} \\ \}. & \end{aligned}$$

The DFA M' equivalent to M has 21 states and is given by
Minimizing M' results in a minimal DFA with 17 states. □

5 Conclusion

We defined a description mechanism for DFAs which lead to a hierarchy of succinct descriptions. We showed that the description mechanism is valid for known descriptions (that is, DFAs, NFAs and AFAs), and we showed the existence of a 3-sNFA for which the equivalent minimal DFA has more than 2^{2^n} states.

δ'	a	b	c
$[[[q_1]]]$	$[[[q_2]]]$	$[[[q_2]]]$	$[[[q_2]]]$
$[[[q_2]]]$	$[[[q_1, q_2]]]$	$[[[q_1]], [[q_2]]]$	$[[[q_1], [q_2]]]$
$[[[q_1, q_2]]]$	$[[[q_1, q_2]]]$	$[[[q_2], [q_1, q_2]]]$	$[[[q_2], [q_1, q_2]]]$
$[[[q_2], [q_1, q_2]]]$	$[[[q_1, q_2]]]$	$[[[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1]], [[q_2]]]$	$[[[q_2]], [[q_1, q_2]]]$	$[[[q_1]], [[q_2]]]$	$[[[q_2]], [[q_1], [q_2]]]$
$[[[q_2]], [[q_1, q_2]]]$	$[[[q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_2], [q_1, q_2]]]$
$[[[q_1]], [[q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1]], [[q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_2]], [[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_2], [q_1, q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_2], [q_1, q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1]], [[q_2]], [[q_2], [q_1, q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1], [q_2]]]$	$[[[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]]]$	$[[[q_1], [q_2]]]$
$[[[q_2]], [[q_1], [q_2]]]$	$[[[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2]]]$	$[[[q_1], [q_2]]]$
$[[[q_1], [q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_2]], [[q_1], [q_2]], [[q_1], [q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1], [q_2]], [[q_2], [q_1, q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1]], [[q_2]], [[q_1], [q_2]]]$	$[[[q_2]], [[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2]]]$	$[[[q_2]], [[q_1], [q_2]]]$
$[[[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1, q_2]]]$	$[[[q_2], [q_1, q_2]]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2], [q_1, q_2]]], [[q_1], [q_2], [q_1, q_2]]]$
$[[[q_2], [q_1, q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1], [q_2], [q_1, q_2]]]$	$[[[q_1], [q_2], [q_1, q_2]]]$
$[[[q_1]], [[q_2]], [[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1, q_2]], [[q_2], [q_1, q_2]]]$	$[[[q_1]], [[q_2]], [[q_1], [q_2], [q_1, q_2]]]$	$[[[q_2]], [[q_1], [q_2]], [[q_1], [q_2], [q_1, q_2]]]$.

References

[1] J. A. Brzozowski and E. Leiss, On Equations for Regular Languages, Finite Automata, and Sequential Networks. Theoretical Computer Science **10** (1980) 19–35. 279

[2] A. Chandra, D. C. Kozen and L. J. Stockmeyer, Alternation. Journal of the ACM **28** (1981) 114–133. 274

[3] J. E. Hopcroft and J. D. Ullman, Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Massachusetts, 1979. 278

[4] E. Leiss, Succinct Representation of Regular Languages by Boolean Automata. Theoretical Computer Science **13** (1981) 323–330. 274

[5] E. Leiss, Succinct Representation of Regular Languages by Boolean Automata II. Theoretical Computer Science **38** (1985) 133–136. 282

[6] 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, 188-191. 274

[7] 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* 278

Author Index

Alegria, Iñaki 1
Alonso, Miguel A. 135
Aranzabe, Maxux 1

Barcala, Fco. Mario 135
Bergeron, Anne 13
Bochmann, Gregor v. 27
Boigelot, Bernard 40
Bultan, Tevfik 74

Champarnaud, Jean-Marc 52

Daciuk, Jan 65
Dang, Zhe 74
Dubernard, Jean-Philippe 87
Duchamp, Gérard 52

Ezeiza, Aitzol 1
Ezeiza, Nerea 1

Farré, Jacques 101
Friburger, Nathalie 115

Gaál, Tamás 125
Gálvez, José Fortes 101
Geniet, Dominique 87
Graña, Jorge 135

Hamel, Sylvie 13
Holub, Jan 149
Holzer, Markus 161

Ibarra, Oscar H. 74

Katritzke, Frank 177
Kemmerer, Richard A. 74
Kempe, André 190
Kutrib, Martin 161

Latour, Louis 40

Maurel, Denis 115
Melichar, Bořivoj 202
Mercer, Robert E. 214
Merzenich, Wolfgang 177
Morey, Jim 214
Mourelle, Luiza de Macedo 221

Nedjah, Nadia 221
Neto, João José 234
Noord, Gertjan van 65

Savary, Agata 251
Sedig, Kamran 214
Skryja, Jan 202

Thomas, Michael 177

Urizar, Ruben 1

Velinov, Yuri 261

Wilson, Wayne 214

Zijl, Lynette van 263, 274