On behalf of the Organizing Committee we welcome you to Porto for the 17th International Conference on Implementation and Application of Automata (CIAA 2012).

This booklet includes the program schedule, some useful practical informations, and the descriptions of the software applications that will be demonstrated at the Demo Sessions. The program will include 5 invited talks, 21 regular paper presentations, 7 short paper presentations, and 12 software demonstrations.

We warmly thank all individuals and institutions that made this event possible. We wish all of you a fruitful CIAA 2012 conference and a nice stay at Porto!

Nelma Moreira
Rogério Reis
**Tuesday, July 17**

08:30 – 09:30 Registration
09:30 – 10:00 Welcoming
10:00 – 11:00  
**Invited Talk: In Search of Most Complex Regular Languages** — Janusz Brzozowski.
11:00 – 11:30 Coffee break
11:30 – 12:30  
- *Implementing Computations in Automaton (Semi)groups* — Ines Klimann, Jean Mairesse and Matthieu Picantin  
- *Regular Ideal Languages and Their Boolean Combinations* — Franz Jahn, Manfred Kufleitner and Alexander Lauser
12:30 – 14:00 Lunch  
14:00 – 15:00  
- *A Pushdown Transducer Extension for the OpenFst Library* — Cyril Allauzen and Michael Riley  
- *A Disambiguation Algorithm for Finite Automata and Functional Transducers* — Mehryar Mohri
15:00 – 16:00  
**Invited Talk: In Memoriam Sheng Yu** — Kai Salomaa
16:00 – 16:30 Coffee break
16:30 – 18:00  
- *On the State and Computational Complexity of the Reverse of Acyclic MinimalDFAs* — Galina Jirásková and Tomáš Masopust.  
- *Hyper-Minimization for Deterministic Tree Automata* — Artur Jeż and Andreas Maletti  
- *Restarting Tiling Automata* — Daniel Průša and František Mráz
18:00 – 19:30 Social Program: Reception

**Wednesday, July 18**

09:00 – 09:30  
**Synchronizing Automata on Quasi Eulerian Digraph** — Mikhail Berlinkov
09:30 – 10:30  
**Invited Talk: A Formal Framework for Processes Inspired by the Functioning of Living Cells** — Grzegorz Rozenberg
10:30 – 11:00 Coffee break
11:00 – 12:30  
- *P(l)aying for Synchronization* — Fedor Fominykh and Mikhail Volkov  
- *Synchronizing Automata of Bounded Rank* — Vladimir Gusev  
- *Synchronization of Automata with one Undefined or Ambiguous Transition* — Martyugin Pavel
12:30 – 14:00 Lunch
14:00 – 16:00  
- *Factor and Subsequence Kernels and Signatures of Rational Languages* — Ahmed Amarni and Sylvain Lombardy  
- *The removal of weighted epsilon-transitions* — Sylvain Lombardy and Jacques Sakarovitch  
- *Weighted LTL with Discounting* — Eleni Mandrali  
- *A Fast Suffix Automata Based Algorithm for Exact Online String Matching* — Simone Faro and Thierry Lecroq  
- *Strict Local Testability with Consensus Equals Regularity* — Stefano Crespi Reghizzi and Pierluigi San Pietro
16:00 – 16:30 Coffee break
16:30 – 18:00  
Demos session N.1
Thursday, July 19

09:00 – 09:30  – Automatic Theorem-Proving in Combinatorics on Words — Dane Henshall, Jeffrey Shallit and Daniel Goč
09:30 – 10:30  
  Invited Talk: Typed Linear Algebra for Weighted (probabilistic) automata — José N. Oliveira

10:30 – 11:00 Coffee break

11:00 – 12:30  – Crossing the Syntactic Barrier: Hom-Disequalities for H1-Clauses — Andreas Reuß and Helmut Seidl
  – On the Descriptional Complexity of the Window Size for Deterministic Restarting Automata — Martin Kutrib and Friedrich Otto
  – Weak Inclusion for Recursive XML Types — Joshua Amavi, Jacques Chabin and Pierre Réty

12:30 – 14:00 Lunch

14:00 – 20:00  
  Social Program: — Visit to “Ribeira”

20:00  
  Social Program: Conference Banquet

Friday, July 20

09:00 – 09:30  – How to Synchronize the Heads of a Multitape Automaton — Oscar Ibarra and Nicholas Tran

09:30 – 10:30  
  Invited Talk: Adding Pebbles to Weighted Automata — Paul Gastin

10:30 – 11:00 Coffee break

11:00 – 12:30  – Nominal Automata for Resource Usage Control — Pierpaolo Degano, Gianluigi Ferrari and Gianluca Mezzetti
  – Weighted Nested Word Automata and Logics over Strong Bimonoids — Manfred Droste and Bundit Pibaljommee
  – Cellular Automata on Regular Rooted Trees — Tullio Ceccherini-Silberstein, Michel Coornaert, Francesca Fiorenzi and Zoran Sunic

12:30 – 14:00 Lunch

14:00 – 14:40  
  Business Meeting

14:40 – 16:00  – On Positive TAGED with a Bounded Number of Constraints — Pierre-Cyrille Héam, Vincent Hugot and Olga Kouchnarenkoe
  – Automata with Modulo Counters and Nondeterministic Counter Bounds — Daniel Reidenbach and Markus L. Schmid
  – Multi-Tilde-Bar Expressions and their Derivatives — Pascal Caron, Jean-Marc Champarnaud and Ludovic Mignot
  – SDFA: Series DFA for Memory-Efficient Regular Expression Matching — Tingwen Liu, Yong Sun, Li Guo and Binxing Fang

16:00 – 16:30 Coffee break

16:30 – 18:00  
  Demos session N.2
Practical Information

Venue

CIAA 2012 is held at the Department of Computer Science (DCC) at the Faculty of Sciences of University of Porto (building FC6). The address is R. Campo Alegre, 1021/1055, 4169 – 007 Porto.

Presentations will take place at the Ferreira da Silva Auditorium, that can be directly accessed from the garden surrounding the building.

Two computer labs will be available for CIAA participants at the first floor of the Department of Computer Science (Laboratório 1 and Laboratório 2). You can login in the computers by entering the user CIAA12 (no password is needed). The computers have installed some of the software systems that will be presented at CIAA12 Demo Sessions. Software Demo Sessions will take place at Laboratório 2.

Wireless Access

Wi-Fi is available in the whole building. In the conference room you just have to select the network called fcup-wifi. To get Wi-Fi in the rest of the building, you have to select the network LabCC.

Lunches

The registration fees cover lunches on the four conference days, and you will find a ticket for each day, in the conference bag. Meals will take place at the Círculo Universitário do Porto, which is located next door of the conference venue. The address is Rua do Campo Alegre, 877, 4150-180 Porto. Phone: 226 094 995

Dinners

There are many restaurants near the hotels area. Here are some suggestions.
What to See Nearby

- Botanic Garden
  Rua do Campo Alegre, 1191
  4150-181 Porto
  Phone: +351 226002153
  jardimbotanico.up.pt
- Casa da Música - Music hall
  Avenida da Boavista, 604-610
  4149-071 Porto

Social Events

Reception

A welcome reception will take place on Tuesday July 17, from 18h30m, in the gardens surrounding the Department of Computer Science.

Excursion and Conference Dinner

The excursion is scheduled for July 19th and includes a visit to Ribeira, one of the most typical districts of Porto, and part of the historical centre, declared World Heritage by UNESCO.

Participants will enjoy a guided visit to Sandeman, one of the prestigious Port Wine Cellars, with wine tasting at the end of the visit. The tour explains the production and warehousing of the Port Wine, telling the secrets of the world’s most ancient demarcated wine region.

After some free time to explore the Ribeira district, we go on a panoramic Six Bridges Cruise on the Douro river. The cruise offers wonderful panoramic views of the city, as well as a special way of enjoying its six bridges.

The Conference Dinner will be held at the end of the tour at Restaurant Zsa Zsa. After the dinner, we return to the conference/hotel area by bus.

Timetable

14.30  Transport by bus to Ribeira (departure yard outside the conference room).
15:00  Guided visit to Sandeman, one of the most prestigious Port Wine Cellars, with wine tasting at the end of the visit.
18:00  Six bridges cruise on the Douro river.
19:00  Transport from Ribeira to the conference dinner at restaurant Zsa Zsa.

Contacts

If needed, you may contact:
  Department of Computer Science: +351 220402980
  CIAA12 Organizing committee: ciaa12@dcc.fc.up.pt
Recently the descriptional complexity of formal languages has been object of extensive research. One of the most studied complexity measures for regular languages is the number of states of its minimal automaton, the state complexity of the language. Other measures can be related to other structural components and other models of computation. The complexity of a language operation is the complexity of the resulting language seen as a function of the complexities of the operation arguments. This prolific research gave origin to a multitude of results scattered over some hundreds of articles, with the inevitable lack of unified terminology and notation. This makes it very difficult for an interested researcher to have a global perspective of this field and realize what is the current coverage achieved in order to know where to allocate more research efforts. All these different aspects and the huge number of results obtained, mainly in the last decades, motivates the need of a tool that helps to organize this information. DesCo is a Web based information system where descriptional complexity results can be structurally introduced, queried, and viewed. This is not an easy task as most of the concepts involved are of abstract nature and thus of difficult classification and computational manipulation.

Formal Languages Descriptive Complexity Database

Descriptive complexity studies the measures of complexity of languages and operations. Usually, the descriptive complexity of an object is its shortest description which can be analyzed in the worst or average case. For each measure, it is important to know the size of the smallest representation for a given language and how the size varies when several such representations are combined or transformed.

Let us describe the information that is kept in the database, while introducing some key notions used in the system. Along with the basic information, some additional details, are also kept. For example, data for interacting with the FAdo system (http://fado.dcc.up.pt), such as which FAdo method generates a deterministic automaton, or performs a given operation. Connections with other symbolic manipulation systems can also be incorporated in the future. The most important objects that our system stores are:

Language Classes – A language class is defined with a name and a description. Given a preorder on language classes, a hierarchy can be considered.

Models of Computation – For each computational model, the system stores its name, abbreviation, a description and which FAdo’s object can be used to represent it.

Complexity Measures – A complexity measure of a language is a function of a size of an associated model of computation. We keep a name, a description and the associated model of computation.

Operations – Operations on formal languages include all boolean operations usually defined on sets plus other specific operations. We also consider simulation (conversion) of different models of the same language.

Operational Complexities – The descriptional complexity of an operation for a given measure and a given kind, is a function of the descriptional complexity of its arguments. Usually, an upper bound is obtained by providing an algorithm that, given the models that represent the operands, constructs a model that represent the resulting language. The complexity of the constructed object as function
of the complexities of the operands gives an operational complexity upper bound. To show that an upper bound is tight, a family of languages (one language, for each possible value of the complexity measures) must be given, for each operand, such that the resulting models achieve that bound, if not, then they at least provide a lower bound.

**Language Families** — To show that a certain complexity bound can be reached, examples of language families $L_n$, where $n$ is related to a complexity measure, must be given, e.g. $L_n = \{x \in \{a, b\} | \#_a(x) = 0 \mod n\}$.

**The Web Interface**

DesCo is implemented in Python and uses the following three toolkits: Pylons, the Web-framework; SQLAlchemy, the interface to the database; and MathJax, the mathematical typesetter. The Web interface allows the manipulation and visualization of all the objects previously described. It is comprised of a series of forms meant for inserting, updating and deleting data, but most importantly, to customize database queries. The picture below shows results for the (worst-case) state complexity of some operations on finite languages.

![Finite Language](image)

**Acknowledges** We thank Janusz Brzozowski, Pedro Coutinho, and Eva Maia for their valuable collaboration in populating DesCo database.
FAdo and GUltar: tools for regular languages manipulation

http://fado.dcc.fc.up.pt/

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Porto, Portugal

The FAdo system aims to provide an open source extensible software library for the symbolic manipulation of automata and other models of computation. To allow high-level programming with complex data structures, easy prototyping of algorithms, and portability, are its main features. Our motivation is the theoretical and experimental research, but we have also in mind the construction of a pedagogical tool for teaching automata theory and formal languages. FAdo currently includes most standard operations for the manipulation of regular languages. For the study of practical performance of algorithms and descriptive complexity (average-case analysis) several tools are also available. FAdo is mainly implemented in Python, a high-level object-oriented language with high-level data types and dynamic typing, that ensures a system which is modular, extensible, clearly and easily implemented, and portable. Specialized and optimized data structures and performance critical algorithms may be written in a low-level language like C, and easily interfaced with Python, via the Cython language extension.

Regular languages can be represented by regular expressions (reex) or finite automata, among other formalisms. Finite automata may be deterministic (DFA), non-deterministic (NFA) or generalized (GFA). In FAdo these representations are implemented as Python classes. Elementary regular languages operations as union, intersection, concatenation, complementation, and reverse are implemented for each class. Many combined operations for DFA have specialized algorithms. Several conversions between representations are implemented: NFA → DFA: subset construction; NFA → reex: recursive method; GFA → reex: state elimination, with possible choice of state orderings and several heuristics; reex → NFA: Thompson method, Glushkov method, follow, Brzozowski, and partial derivatives. For DFA several minimization algorithms are available (some with C implementations): Moore, Hopcroft, Brzozowski, and some incremental algorithms. Some support is provided for computing syntactic semigroups. There are several algorithms for language equivalence. Finite languages (FL) can also be represented by tries and AFA (acyclic finite automata). FAdo provides an interface to the GraiL+ system [8] that allows the execution of any GraiL+ command and the import/export GraiL+ file formats.

Exact and random generators for some classes of automata and regular expressions were designed and implemented: an exact and a uniform random generator are available for initially connected DFA (ICDFA) [3]; exact enumeration of (minimal) acyclic DFA based on canonical forms [4]; uniform random generators for several classes of regular expressions and sizes; uniform random generators for tries; and non-uniform random generators for NFA. In order to have a reasonable sized (enough for statistically significant results), consistent, random sample readily available, we designed and implemented three relational databases to store datasets of random ICDFA, NFA, and reex. In general, each dataset has 20 000 objects of the same size and alphabet. For more details on the models and the respective operations implemented see the documentation file http://fado.dcc.fc.up.pt/resources/Files/FAdo.pdf.
GUItar is a visualization software tool for various types of automata. Its purposes include automatic and assisted diagram drawing, algorithm animation, interactive editing and export/import filters. After a first version implemented in wxPython [1, 2] we are now redesigning GUItar with the Qt graphical framework which guarantees a solid foundation for an efficient, extensible and cross-platform application architecture. GUItar provides a generic interface between the diagram graphical system and external modules, such as automata manipulation tools (e.g. FAdo) or typesetting graphical packages (e.g. VauCanSon-G).

GUItarXML [5] is a XML format that was designed for the description of diagrams and that allows several information layers. It is compatible with GraphML [6] and borrows some functionalities from the FSMXML format [7].

References
The New Grail+

Cezar Câmpeanu\textsuperscript{1}, Yuan Gao\textsuperscript{2}, Xu Han\textsuperscript{1}, Lila Kari\textsuperscript{2}, Andrei Păun\textsuperscript{3,4}, and Rui Zhou\textsuperscript{1}

\textsuperscript{1} The University of Prince Edward Island  
\textsuperscript{2} Western University  
\textsuperscript{3} Lousiana Tech University  
\textsuperscript{4} University of Bucharest

1 About Grail+

There are several packages dealing with formal language objects, many of them earning reputation during a long period of time. We list here just a few, in no particular order: Grail+, Vaucanson, FaDo, FSA, libfa and so on.

Grail is now developed at the Computer Science Department at Western University\textsuperscript{5}, London, Ontario, Canada. Grail+ is intended for use in teaching and research. The Grail+ project was until recently coordinated by Dr. Sheng Yu, professor of Computer Science at Western University.

The original work on the Grail project was done by Darrell Raymond and Derick Wood, and it was published as technical reports in:

- Computer Science Technical Report; HKUST-CS95-17 The Grail papers: version 2.3
- CS-93-01, Department of Computer Science, University of Waterloo (January 1993) Grail: Engineering Automata in C++
- TR #345, Department of Computer Science, University of Western Ontario (1993),


2 What is new in Grail+ 3.4

The latest "known" version of Grail+ was 3.0 and was released in 1998. Several bugs were fixed and many improvements were done during 1998 and 2001, mainly at The University of Western Ontario under the direct supervision of Sheng Yu.

It must be noted that Grail project was started long before C++ was standardized (1998), also before standard library was written. With the increasing development and popularity of Linux and GNU tools, it was desirable to make Grail+ available for more platforms. Most of the code was not obeying to the new standard, so it was necessary to change the code to a portable format that would easily compile on most platforms. In 2003, Andrei Păun ported Grail 2.5 for the GNU C++. His version could also compile with the VC++ from Microsoft, without changing anything. Since Grail 3 had more functions implemented, it was desirable to add them to the new code. The work was carried at Louisiana Tech University under the direct supervision of Andrei Păun. During this time, at The University of Prince Edward Island new functions were added to Grail+, most notably being the shuffle and the inverse shuffle operations; cover automata support was added at Louisiana Tech University.

Because some work was independently done, a few years ago we discovered that there are parallel development branches.

Grail 3.4 includes the latest verified code from all previous branches. This work was done by Xu Han and Rui Zhou, under the direct supervision of Cezar Câmpeanu.

\textsuperscript{5} formerly The University of Western Ontario
Also, the bugs reported in various venues were fixed, most notably being the one related to non-canonical numbering of states, and the bug in fmenum[1].

Debugging feature, i.e., displaying internal work of the algorithm, was also extended, mainly for cover automata algorithms.

We emphasize here some of the new filters added to version 3.4, versus version 3.0:

- fltofmmn: finite language to minimal DFA;
- fltofcm: finite language to minimal DFCA;
- fmtofcm: DFA to minimal DFCA Körner/Goeman algorithm;
- fmtofcm2: DFA to minimal DFCA gap algorithm;
- fmtofcm0: the original minimization algorithm for DFCA;
- flsimset: shows similarity sets;
- fmsubclose: subword closure;
- fltoint: one to one function transforming a finite language into a natural number providing a natural Gödelization;
- inttofl: inverse of fltoint;
- fmshuffle: computes the shuffle of two automata;
- fmshuffleq: computes the inverse shuffle of two automata;
- fmminis: the Moore minimization algorithm for DFA.

References

1. Ackerman, M., Shallit, J.: Efficient enumeration of words in regular languages. Theoretical Computer Science 410(37), 3461–3470 (September 2009)
I-LaSer: Independent Language Server\textsuperscript{1}

Stavros Konstantinidis and Meng Yang
Department of Mathematics and Computing Science, Saint Mary’s University,
Halifax, Nova Scotia, B3H 3C3 Canada
s.konstantinidis@smu.ca, meyang.mike@gmail.com

1 Introduction

I-Laser is a software consisting of a web interface [8] and an implementation of several algorithms written in C++ and PHP. These algorithms make extensive use of the FAdo library for finite state machines [1, 5]. I-Laser is currently capable of answering the following decision problem.

– The satisfaction question: Given the description of a regular language and the description of an independence property, decide whether the language satisfies the property.

The web interface allows the user to specify the files containing the descriptions of the regular language and the desired property. Regular languages can be described via NFAs using the Grail [6] or the FAdo format [5]. The issue of describing language properties is a central theme in this work. We provide a summary in the next sections. The reader is referred to [4] for details.

2 Independence Properties

Reference [7] provides details about the general concept of independent language properties. Here we restrict our attention to 3-independence properties, or equivalently, properties defined by binary relations [10]. Let $\rho$ be a binary relation over the set of words $\Sigma^*$. A language $L$ is $\rho$-independent if no words $u, v \in L$ are related via $\rho$; that is, neither $(u, v) \in \rho$ nor $(v, u) \in \rho$ holds. If $\mathcal{P}_\rho$ is the set of all such languages (the property defined by $\rho$) then we say that $L$ satisfies $\mathcal{P}_\rho$. Examples of independence properties include prefix, suffix, bifix, infix, outfix, and solid codes, hypercodes, error-detecting languages for various cases of error situations.

3 Describing Independence Properties

In [3], the author provides a method for defining independence properties via sets of trajectories. In particular, a formal method for describing an independence property is via a regular expression $e$ over \{0, 1\} such that a language $L$ satisfies the property if

$$L \cap (L \sqcap_e \Sigma^*) = \emptyset,$$

where $\sqcap_e$ is the shuffle operation on the trajectory set $e$. Users can enter regular expressions in I-LaSer using the FAdo format. For example, $0*1*$ describes the prefix code property and $0*1*0*$ the infix code property.

In [4], the authors provide a formal method for describing independence properties via transducers. More specifically, two types of properties are considered. In the first one, called the input-altering transducer property, a property is defined via an input-altering transducer $t$ such that a language $L$ satisfies the property if

$$L \cap t(L) = \emptyset,$$

\textsuperscript{1} Research supported by NSERC.
where a transducer $t$ is called input-altering if $w \notin t(w)$, for any word $w$. For example, if $t$ is a transducer that, on input $w$, outputs all proper prefixes of $w$, then $t$ describes the property of prefix codes. On the other hand, if, on input $w$, $t$ outputs all proper infixes (subwords) of $w$, then $t$ describes the property of infix codes. Users can create a text file containing the description of $t$ using FAdo-like, or Grail-like format. In [4] it is shown that this method can be used to express all properties described by regular sets of trajectories.

The second type of property is called the input-preserving transducer property such that a language $L$ satisfies the property if, for all $x \in L$,

$$\left((L - x) \cap t(x) = \emptyset, \right)$$

where a transducer $t$ is called input-preserving if $w \notin t(w)$, for any word $w$. This method is more expressive than the previous one and can be used describe the property of error detection [4].

4 Answering the Satisfaction Question

I-LaSer provides a web interface where users can describe a desired property via a regular expression (trajectory set), an input-altering transducer, or an input-preserving transducer. In each case, the software tests the corresponding condition (1), (2), or (3) shown above. Condition (2) can be tested using standard product constructions involving NFAs and transducers, and the emptiness test for NFAs. For condition (1), I-LaSer converts the given regular expression to an input-altering transducer such that (1) holds if and only if (2) does. For testing condition (3), we construct a real-time transducer $s$ [9] that depends on the given input-preserving transducer $t$ and the given regular language such that (3) holds if and only if the transducer $s$ is functional. For testing functionality, we use the algorithm in [2].

References

**Abstract.** libFAUDES implements data structures and algorithms for finite automata and regular languages. The library takes a control theoretic perspective as originally introduced by P.J. Ramadge and W.M. Wonham in the 1980’s and, since then, actively discussed within the scientific community. **libFAUDES** is an open project, contributions to the further development are highly appreciated.

**Supervisory Control**

For a concise introduction, consider the so called *plant* to be modelled by a prefix-closed language $L$ over $\Sigma$. Here, $L$ is interpreted as the set of all possible strings the plant can exhibit. Furthermore, assume another prefix-closed language $E \subseteq L$ to represent a formal *specification*. Here, $E$ is interpreted as the set of all acceptable strings. In the situation where $E$ is a strict subset of $L$, the plant may exhibit unacceptable strings, and one asks for a *supervisor* that interactively restricts the plant behaviour accordingly. For this purpose, the alphabet $\Sigma$ is decomposed as a disjoint union of *controllable events* and *uncontrollable events*, $\Sigma = \Sigma_c \cup \Sigma_{uc}$, respectively. The supervisor may at any instance of time disable any controllable event. Technically, this is expressed by the *controllability condition* $K \Sigma_{uc} \cap L \subseteq K$ imposed on the closed-loop behaviour $K \subseteq L$. A crucial observation is that, given a plant and a specification, there uniquely exists a supremal controllable sublanguage $K^+ \subseteq E \subseteq L$. Moreover, if $L$ and $E$ are regular, then so is $K^+$, and a realisation $S$ can be computed from automata representations of $L$ and $E$. Here, $S$ is interpreted as an implementation of a supervisor that, when operating the plant, enforces the specification by strategically disabling controllable events.

Since the original work of P.J. Ramadge and W.M. Wonham in the 1980’s, many researchers have contributed to supervisory control. This includes further development of the theoretical framework, e.g., to cover hierarchical, modular and decentralized control architectures, as well as discussions on implementation issues, e.g., automatic code generation for Programmable Logic Controllers (PLCs) in the context of industrial applications. However, there are still various open questions and, hence, supervisory control remains an active field of research.

**Scope of libFAUDES**

The main purpose of **libFAUDES** is to promote the framework of supervisory control theory in two regards. First, **libFAUDES** implements a wide range of algorithms on finite automata, including general functions (e.g., boolean operations on the associated languages), solutions to common controller synthesis problems (e.g., supremal controllable and normal sublanguages), simulations to inspect and debug closed-loop dynamics and the execution of controllers on the actual physical plant (so called *hardware-in-the-loop simulation*). Thus, **libFAUDES** can serve as an environment to evaluate a design method in a real-world application context. The second aspect in the design of **libFAUDES** is to provide a software infrastructure to reduce the coding effort for newly developed methods in supervisory control.

Implementation details:

**C++ Library.** **libFAUDES** is written in C++ with use of the Standard Template Library (STL). The generator classes are set-based models that resemble the definition of a quintuple automaton $G = (Q, \Sigma, \delta, Q_o, Q_m)$. Using **libFAUDES** to implement algorithms originally stated in terms of automata is
straightforward and benefits from the general infrastructure, like file I/O and graph visualisation via Graphviz/dot\(^1\).

Non-restrictive Open-Source License. libFAUDES sources are distributed for free under conditions of the GNU Lesser General Public License (LGPL). There are no relevant restrictions to use the library in non-open source and/or commercial projects. libFAUDES comes with a (compile-time-) plug-in mechanism to extend the library in an organised manner. In particular, plug-in developers are invited but not forced to contribute their code to libFAUDES.

Lua Scripting Support. The interpreter luafaudes makes libFAUDES data types and functions available within the scripting language Lua\(^2\). The wrapper code that integrates libFAUDES with Lua is automatically generated by the build-system. The additional coding effort required to provide access to library extensions from within Lua is minimal.

User Reference. The libFAUDES build system can organize an optional user reference, to complement the C++ API documentation. Thus, developers who wish to illustrate, motivate or explain their algorithms by examples, informative text or formal definitions, can directly address users of libFAUDES based applications, e.g. luafaudes or the graphical front-end DESTool.

Further Information

libFAUDES is currently maintained by the Lehrstuhl für Regelungstechnik, University Erlangen-Nürnberg, Germany. A detailed user reference, examples, literature references, a list of contributing authors, and downloadable packages are available at the project web page www.rt.eei.uni-erlangen.de/FGdes/faudes.

This summary for the presentation of libFAUDES at CIAA12 has been composed by Thomas Moor, Lehrstuhl für Regelungstechnik, University Erlangen-Nürnberg, Germany, Klaus Schmidt, Department of Mechatronics Engineering, Çankaya University, Turkey, and Tomáš Masopust, Institute of Mathematics, Academy of Sciences, Czech Republic.

---

\(^1\) See www.graphviz.org
\(^2\) See www.lua.org
LISA: a new programming language for finite automata and other formal language objects

Cezar Câmpeanu and Seyed Pooria Madani Kochak
The University of Prince Edward Island

1 What is LISA?

LISA is an acronym for Language Interpreter and Structural Analyzer and is a new programming language designed to speed up writing tests for automata and other formal language objects. Thus, writing programs to test several operations on regular languages can be programmed much faster and can be easily maintained. The project was started as an undergraduate honours thesis by Pooria Madani, under the supervision of Cezar Câmpeanu, at The University of Prince Edward Island.

For the moment, LISA is using GRAIL+ as a back-end engine, but the compiler can be adapted to accept other back-end engines, like MiniGrail/MachineCat, Vaucanson, or FSA.

It must be noted that LISA compiler produces native code, thus it is much faster to use this programming language to write tests, versus writing scripts that perform the same tasks.

It is probably slower than writing programs using Grail+ package as a library, but in this case, it requires a deep knowledge of the internal structure of the Grail+ package.

Thus, researchers will only need to learn one uniform programming language syntax and will have the liberty of using a variety of software packages as the back-end computational engine.

When completely implemented, various back-end engines can be tested within LISA programs, thus, newly implemented algorithms can be tested in a much uniform way.

2 Programming in LISA and Generating ICDFA

In [1] is described an uniform way of generating non-isomorphic Initially Connected Deterministic Finite Automata, method that is very useful in testing the performance of various algorithms.

In LISA we have two methods for generating ICDFAs:

– exhaustive generation of DFCAs and
– random generation of DFCAs,

both methods using the algorithms described in [1].

We show, for the simplicity of the presentation, a sample code written is LISA:

```c
// Enumerate all the possible ICDFA n=3, k=2;
// compute the union between all of two ICDFAs
// and find the two ICDFA that generate largest DFA;
// display the result.
declare{
    int Max;
    int m; int n;
    dfa MaxA; dfa MaxB;
    dfa A; dfa B; dfa C;
}
```

Acknowledgments to Chris Vessey, who suggested the name.
program{
    Max = 0;
    generate(enumerate,3,2){
        A = next;
        generate(enumerate,3,2){
            B = next;
            C = union(A,B);
            C = reduce(C);
            if(!iscomplete(C)){
                C = complete(C);
            }
            m = size(C);
            if(m > Max){
                Max = m;
                MaxA = A;
                MaxB = B;
            }
        }
    }
    print("The size of union is: ", Max);
    print(MaxA);
    print(MaxB);
    print(union(MaxA,MaxB));
}

In the above example, replacing keyword \texttt{enumerate} by \texttt{random}, we obtain a random enumeration, and to exit the loop, a \texttt{break} instruction must be used.

These kind of programs can be used for evaluating the state complexity of various operations. Unfortunately, although isomorphic automata are not generated, in case of exhaustive tests involving embedded loops for automata with more than 5 states, the number of objects generated is huge, so the running time increases significantly.

At this time, most of the filters implemented in Grail+ are also natively supported by LISA.

References

A New Software Package: MiniGrail*

Cezar Câmpeanu and Rui Zhou
The University of Prince Edward Island

1 Why a new package?

The goal of this project was to produce a better structure for a software package that has the same functionality of Grail+. Although there are several packages used for symbolic computation, Grail+ has two major advantages over the other ones:

– it accepts input and produces output in a human readable format;
– its code has been extensively tested by many people in the Formal Language and Automata research community, and all reported bugs have been fixed (as of version 3.4).

The main disadvantages of Grail+ are:

– Most of the Grail code was written before the standardization of C++. Many of the classes used by Grail+ conflict with C++ Standard Template Library (STL).
– For Grail+, algorithms are member functions of a given class. Therefore, if an algorithm uses two objects from two different classes, we must modify the code for both classes. Such cross references are relatively frequent, thus, adding a new algorithm, or implementing a new filter must be done with extreme care, since all cross references must be checked.
– The main program is huge and the code is not elegant. Adding new filters, although it can be easily done, requires an extensive alteration of the code in several locations of the package tree.

Hence, the primary goal of the new package was to solve exactly these problems, while maintaining all the other features of Grail+. The secondary goal was to keep, if not improve, the amount of resources used in computation. Thus, the memory used during computations and runtime should be similar, or even better than the ones used by Grail+.

2 Machine Cat

The project was started under the name of MachineCat as an undergraduate honours thesis by Rui(Ray) Zhou, under the supervision of Cezar Câmpeanu, at The University of Prince Edward Island.

Since, new features and functionality have been added to the project, presently, more than half of the filters implemented in Grail+ are also present in MiniGrail.

Here we emphasize some features present in MachineCat (also inherited in MiniGrail):

– MachineCat’s main function is very short, because all the filters and algorithms are stored in arrays of functions. The Makefile uses a script that automatically generates new designated header files, which changes the content of the function arrays. These designated header files are included in the main function by default, thus the main function is not changed for every new algorithm added to the package.
– The project structure is organized following a very strict hierarchy, without (or very little) circular dependencies.
– Adding filters and algorithms is done automatically. Just write the code and put it in the right folder; the system will take care of the rest, more exactly, to include it properly into the software package.
– Porting the code from Grail+ to MiniGrail can be done with little effort.
3 Performance tests

All performance tests, done by generating automata with the number of states in the order of tens of thousands, show that MiniGrail is as good, or even better than Grail+. Apparently, for most of the tests, the input system uses most of the time resources. Thus, MiniGrail reads automata using a finite transducer that converts on the fly the input into the internal format. Moreover, in case of a noncannonical numbering of states of a finite machine, the system will allocate the minimum amount of memory by converting all symbols into (large) integers stored as sequences of bytes, using two’s complement code. Thus, the input system for MiniGrail is much faster than the one of Grail+. the only overhead is the symbol table that must be stored internally. However, this approach can help us in displaying the internal work of the algorithms. This feature is difficult to implement in Grail+.

4 Future Development

Currently, all the development and tests were carried out in Linux environment, but as MiniGrail is developed with standard C++ and its Standard Template Library, it should be easily transferred to other platforms, with little modification.
PAPAGENO: a Parallel Parsers Generator for Operator grammars

Alessandro Barenghi  Ermes Viviani
Stefano Crespi Reghizzi  Dino Mandrioli  Matteo Pradella
Dipartimento di Elettronica e Informazione - Politecnico di Milano
{barenghi,viviani,crespi,mandrioli,pradella}@elet.polimi.it

Abstract

In almost all applications, languages are parsed employing traditional algorithms (mostly deterministic such as LR(1) parsers generated by Bison), which are sequential due to their left-to-right state-dependent flow. Although early theoretical studies (see Appendix 1) on parallel parsing algorithms delineated the possible speedup that could be obtained in principle on abstract parallel machines using a data-parallel approach, practical developments have not materialized, except in recent years on ad hoc parsers for XML or HTML browsing. PAPAGENO is a new tool generating parallel deterministic parsers that offer significant speedups on large input texts, while not penalizing performances on shorter ones. The generated parser splits the source text into arbitrary chunks that are parsed in parallel and then recombined into the whole parse tree. We use Floyd’s operator precedence grammars (see Appendix 2), a precursor of LR(1) grammars, which are intrinsically more amenable to parallelization, yet are expressive enough to specify most real languages. To recombine parsed chunks, we have experimented different approaches using either single or multiple workers. On both synthetically generated texts and large real documents, our parsers exhibits significant speedups on multi-core/processor machines. The experiments show that the code portion executed in parallel is above 80% for most scenarios. For large real documents the results scale well up to 16 processors. On single processors our parser is slightly faster than state-of-the-art parsers generated by Bison.

Tool Components and Demo Outline

PAPAGENO (Parallel Parser Generator for Operator precedence grammars) has a structure similar to other parser generator tools, taking as input a specification of the language grammar and producing as output an implementation of the parser for the language. The generated parser, coupled with a scanner, is able to perform any action specified by the user in correspondence of a reduction. PAPAGENO takes as input a context-free (BNF) grammar represented in the format used by Bison for the syntax layer, and outputs the parallel parser code, with suitable interfaces to lexical analysers (such as the ones generated with Flex). PAPAGENO can be used as a drop-in replacement for Bison whenever a grammar compliant with Floyd’s conditions is available. The given grammar is screened against Floyd’s conditions, in particular that it is in operator form and has no precedence conflicts and no repeated right hand sides.

The innovative and critical part of the tool concerns the parallel parser structure: it is organized as a configurable set of identical parallel threads, each one acting as a worker to parse up to saturation a chunk (substring) of the input text. The number of threads is a parameter to be specified at runtime when calling the parsing routine, thus allowing to choose how many workers are active, without the need to re-generate the parser. Every parsing worker parses as far as possible its chunk, producing a pushdown stack and a (partial) Abstract Syntax Tree (AST). The partial pushdown stacks and ASTs are combined together by one or more threads, which bring the parsing process to completion. To enhance performances the parser is written in C and incorporates several optimizations, such as an efficient representation of both the precedence matrix and the grammar rules employed in the reductions. In addition to the parser generator, the tool offers a typeset output of the precedence matrix and diagnostic messages to guide the developer when correcting the causes of possible precedence conflicts.

Experimental Results

Parsers will be generated for different grammars, such as the canonical one for arithmetic expressions and the grammar of JSON, the common lightweight format for data interchange employed in a vast number of web applications.

1 Work partially supported by Google Research Award.
Fig. 1. Speedups with different string lengths (80k-10M tokens, lighter lines are shorter strings), and running time comparison against Bison on a 16 core platform.

**Demo steps**

1. The tool is obtained via download from a public repository.
2. Source language specifications for two grammars are provided: the common arithmetic expressions grammar, and the JSON grammar. Since the official JSON grammar is not natively in operator form, a minor modification is performed to cope with this problem.
3. The input texts are randomly generated arithmetic expressions for the first grammar, and real world JSON datasets such as the configuration data from AdBlocker (80 kB), and a file containing statistics on n-grams present in English in Google Books (10 MB).
4. Running the parser on a multi-core notebook
   - (a) using 1 core only: The performances of the PAPAGENO generated parser will be compared against the one of a Bison generated one on a single core scenario
   - (b) using all the available cores: The speedups attainable with PAPAGENO on an off-the-shelf hardware platform will be shown on the available laptop.

The appendices cover some related research and the basic definition of Floyd grammars. Further developments of the tool are planned to cope with incremental parsing and to enlarge the class of grammars beyond FG.

*Acknowledgements* To Brad Chen of Google Inc. for practical indications on the benchmarks, and in particular for the suggestion on the use of JSON. To Valerio Ponte for participation to the first implementation.
The Spot Library, and its Online Translator from LTL (and PSL) to Büchi Automata

http://spot.lip6.fr/

Alexandre.Duret-aLutz@lrde.epita.fr

1 Introduction

Spot [12] is a C++ library of algorithms to implement the automata-theoretic approach to model-checking of linear-time temporal properties. To verify that a property $\varphi$ holds on a model $M$ of some system, this approach [19] splits the verification process into four operations:

1. Computation of the state-space for the model $M$. This state-space can be seen as an $\omega$-automaton $A_M$ whose language, $L(A_M)$, represents all possible infinite executions of $M$.

2. Translation of the temporal property $\varphi$ into an $\omega$-automaton $A_{\neg \varphi}$ whose language, $L(A_{\neg \varphi})$, is the set of all infinite executions that would invalidate $\varphi$.

3. Synchronization of these automata. This constructs a product automaton $A_M \times A_{\neg \varphi}$ whose language, $L(A_M \times A_{\neg \varphi})$, is the set of executions of $M$ invalidating $\varphi$.

4. Emptiness check of this product. This decides whether $A_M \times A_{\neg \varphi}$ accepts an infinite word, and can return such a word (a counterexample) if it does. The model $M$ verifies $\varphi$ iff $L(A_M \times A_{\neg \varphi}) = \emptyset$.

While Büchi Automata (BA) are commonly used in such a framework, Spot implements a generalization thereof called TGBA (Transition-based Generalized Büchi Automata) where infinite words are accepted iff they visit infinitely often one transition from each acceptance set. TGBA are as expressive as BA, but can be more compact than BA, especially when expressing weak fairness properties.

2 LTL Translation

<table>
<thead>
<tr>
<th>Count of nondeterministic states and automata</th>
<th>Products with a random state-space of 200 states</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma</td>
<td>A_{\neg \varphi}</td>
</tr>
<tr>
<td>nondet. st. tr. st. aut. st. tr.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool</th>
<th>Cumulated sizes of automata</th>
<th>Spot 6.1.0 (7 timeouts)</th>
<th>ltl2ba 1.1</th>
<th>LTL→NBA</th>
<th>Modella 1.5.9</th>
<th>ltl3ba 1.0.1</th>
<th>Spot 0.9.1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>188 formulas from the litterature</td>
<td>1 635 7 825 1402 176 314 218 21 549 478</td>
<td>1 080 3 646 871 177 215 717 12 766 425</td>
<td>989 3 214 784 178 197 568 12 063 463</td>
<td>1 391 4 562 679 125 274 281 10 907 038</td>
<td>884 2 538 349 126 175 626 6 562 700</td>
<td>742 2 018 139 51 147 709 5 396 354</td>
</tr>
</tbody>
</table>

Spot is renowned for its translation of Linear-time Temporal Logic (LTL) formulas into small BA (or TGBA). The above table compares the BA output by Spot to those output by some other tools and shows that the automata output by Spot are, on the average, smaller, more deterministic, and yield substantially smaller products (in all columns, smaller numbers are better). More detailed benchmarks are available at http://spot.lip6.fr/dl/bench-0.9.1.pdf.

Spot obtains small automata thanks to a combination of: many syntactic simplifications on LTL formulas, a tableau construction from LTL to TGBA that uses binary decision diagrams for many simplifications [10], several post-processing simplifications including a minimization algorithm applied to weak deterministic Büchi automata [9] and a simulation algorithm applied to TGBA that are not weak deterministic.

Spot’s LTL translator has been used in a number of benchmarks [16, 15, 7, 3], and as the translation step of several approaches [17, 18, 13, 6]. Spot’s translator can be tested and easily demonstrated on-line at http://spot.lip6.fr/ltl2tgba.html. This web page offers access to three of the four translation algorithms implemented in Spot, with options to enable/disable all pre- and post-processings, and a choice of different automaton output (BA, TGBA, monitor). This interface can also be use to classify LTL formulas into the Manna-Pnueli hierarchy [14], either syntactically, or (for the obligation, safety, and guarantee classes), structurally.
Recent versions of Spot have added support for the linear fragment of PSL (Property Specification Language) an industrial standard \cite{psl1} that mixes LTL operators with extended regular expressions. This can also be demonstrated online. Besides being a showcase for a subset of the features available in the Spot library, we have found this online translator to be a great help in our day-to-day research on LTL/PSL model-checking.

3 Model-checking and More

Amusingly, Spot’s main purpose, as a library, is not really to translate temporal logic into automata: it is to help people build model checkers for their custom formalisms. All they need is to create an object that follows the TGBA interface and that performs the on-the-fly computation of the state-space of their models. TGBA are abstract objects in Spot, and they have an interface that allows on-the-fly computations: this way we build only the part of the state-space, and the part of its product with the property automaton, that the emptiness check needs to explore. We have built at least six custom model-checkers using Spot (four of which are for different flavors Petri Net models).

Spot actually offers more than just algorithms to implement the above automata-theoretic approach. It has four algorithms to translate Linear-time Temporal Logic (LTL) to TGBA or BA, five emptiness checks algorithms with some variants, two Büchi complementation algorithms, several LTL simplification rewritings, and a couple of reduction algorithms for TGBA, etc.

Spot has been used to experiment with various components of the standard approach \cite{psl1,babiak2003}, but also as an \( \omega \)-automaton library to implement non-model-checking procedures \cite{psl2}.

Spot is developed collaboratively between LRDE (www.lrde.epita.fr) and LIP6 (move.lip6.fr). Its source code is distributed under the GNU GPL at spot.lip6.fr, and comes with a large test-suite.

Comments, questions, and bug-reports can be sent to our mailing list at spot@lrde.epita.fr.

References

\begin{thebibliography}{11}
\end{thebibliography}
The T3S tool was developed by Pedro et al. [2,3,1] and implements the algorithm developed and described in these references. The tool allows the representation of stochastic timed automata (STA), it is able to learn STA, and also generate an event-based model that can be used as input for a statistical model checker Ymer [4]. Moreover, this tool is extended and developed as a Matlab toolbox, namely the T3S Matlab Toolbox.

The T3S Matlab toolbox\(^4\) provides an intuitive and simple graphical user interface (GUI) as depicted by Fig. 1. The GUI was developed to bridge the gap of graphical representation for automata-based models in Matlab and is a diagram designer for Markov chains. It has been developed with the Qt toolkit and contains an API that enables the tool to be integrated with other applications.

The T3S tool is able to model and learn systems that do not hold the Markov property such as the generalized semi-Markov process (GSMP). Such processes cover a wide range of stochastic systems, examples include land-satellite communication systems and multi-processor system scheduling with uncertainty in task duration.

The logic architecture of the learning process supported by the tool is depicted by the diagram in Fig. 2. The transitions represent the applied methods, and the locations represent the output model or the data structures provided by the applied methods. The learning process starts by considering a set of sample executions that will drive the learning process, note that the T3S tool can generate sample executions from a known GSMP. The resulting simulation method is also able to generate sample executions that can be used to learn, and therefore the TS3 tool is able to create a Closed-Loop learning system as illustrated in Fig. 2. The main ingredients for the learning algorithm are the Schedule Estimator (SE), the Probabilistic Similarity of States (PSS), and the Model Selection (MS). The SE receives an event-driven prefix tree constructed from sample executions, and produces as output an event-driven prefix tree with the clocks changed (to original samples). The PSS uses its data structure to establish a stable relation between states. The event-driven prefix tree is assumed as a particular case of a stochastic automaton that allows the merging of equivalent states. As output it produces a STA. Meanwhile, the

---

\(^4\) The T3S Matlab toolbox v1.0 runs on Matlab 2010b and needs to be compiled with gcc version 4.3 and Qt 4.7. A standalone binary of the T3S tool is also provided for Windows and Linux at http://t3stool.sourceforge.net.
model selection is applied in order to estimate the parameters of distributions for each event. Lastly, from this information the toolbox is able to produce a complete GSMP that recognizes (in the limit) the same language of the learnt system.

References

3. de Matos Pedro, A., de Sousa, S.M.: Learning generalized semi-markov processes: From stochastic discrete event systems to testing and verification. Technical Report DCC-2012-01, Department of Computer Science, University of Porto (March 2012), this is an expanded version of [2].
Vaucanson 1.4.1

VAUCANSON is an on-going project of a free software platform dedicated to the manipulation of finite state automata. ‘Finite state automata’ is to be understood in the broadest sense: VAUCANSON supports weighted automata over a free monoid, and weighted transducers, that is, weighted automata over products of two free monoids.

From user’s point of view, the platform consists in two main components:

The VAUCANSON library is a C++ library that implements objects for automata, rational expressions, as well as algorithms on these objects. This library is written under the static genericity paradigm.

TAF-Kit is a command-line interface to the library that allows user to execute VAUCANSON’s algorithms without any knowledge of C++ nor of VAUCANSON API. This interface is instantiated for a predefined set of commonly used automaton types.

The platform is coupled with two other communication modules:

An XML format for automata and expressions, called FSM XML, aims at being a general purpose interchange format for weighted automata and regular expressions. It is used as the normal, and default, input and output format for TAF-Kit and thus for the communication between TAF-Kit and Vgi.

A Graphical User Interface called Vgi and especially dedicated to VAUCANSON, allows to describe automata and to visualize the result of operations on automata in an interactive and graphical way. All functions defined in TAF-Kit can be called via the menu of Vgi.

The VAUCANSON platform is currently under a thorough revision and its core is rebuilt with a new design that changes the interface and the associated API. The ultimate version of the first phase of the project, coined VAUCANSON 1.4, has been presented at FSMNLP 2011, and eventually released in September 2011. It is accessible through the TAF-Kit only, which is also the only part that is documented.

The novelty of the VAUCANSON 1.4.1 version, just released in July 2012, is the integration of the Graphical User Interface Vgi. Vgi is designed to provide a sophisticated yet user-friendly interface for the user to interact with the platform more effectively.

The VAUCANSON platform is designed and realised by VAUCANSON GROUP. This work is supported by the ANR-10-INTB-0203 project since March 1st, 2011.

The Graphical User Interface Vgi is designed and realised at the Computation Theory Laboratory, EE Dept., National Taiwan University in Taipieh. It is supported by the NSC-100-2923-E-002-001-MY3 project since January 1st, 2011.
VAUCANSON 1.4.1 and Vgi can be downloaded from the VAUCANSON PROJECT site:

http://www.vaucanson-project.org

where all relevant informations for the installation, as well as complete TAF-Kit and Vgi user manuals, are to be found.

TAF-Kit stands for Typed Automata Function Kit. In the static generic programming paradigm used in the VAUCANSON library, the types of the automata that are treated have to be known at compile time. The command-line interface TAF-Kit is a set of programs called from the shell and used to chain operations on automata. At the installation of VAUCANSON, TAF-Kit is therefore compiled for several predefined types of automata.

TAF-Kit does not allow to write new algorithms nor to manipulate new types of automata, but it makes it possible, without efforts, to use already programmed functions on automata of predefined types. TAF-Kit gives a restricted access to VAUCANSON functionalities, but it is a direct access, without any need of programming skill. A basic familiarity with Unix command syntax only is sufficient to make use of TAF-Kit.

The type of an automaton is determined by: (a) the fact it is an automaton over a free monoid, or over a product of two free monoids, (b) the type of the generators of the free monoid(s), (c) the weight semiring. The weight semiring can be \( \mathbb{B}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{F}_2, \mathbb{Z}_{\min}, \) or \( \mathbb{Z}_{\max} \). The generators may be characters, integers or pairs of these. At the installation of VAUCANSON, TAF-Kit is instantiated for 18 combinations of weights and generators types.

Besides basic editing commands, most of ‘classical’ operations on automata, together with less classical ones, are available in the TAF-Kit instances: from transformations of automata into expressions and back to operations such as quotient, product, and shuffle, from operations for Boolean automata such as determinisation or computation of the universal automaton to operations on transducers such as composition or evaluation, from reduction of automata with weights in a field to transformation of automata over pairs into transducers.

The architectural design of Vgi comprises of a hierarchy of layers, making it robust, extendable and portable. In addition to those basic drawing functionalities existing in typical graphical user interfaces, the Vgi allows the user to read/store geometric information associated with layouts of automata from/to their external representations kept in XML files, as well as to call TAF-Kit commands directly. Advanced visualization techniques such as semantics-based layouts and focus+context display and navigation are being developed, hoping to reveal more insight into the evolution of an automaton in the process of the application of algorithms.

VAUCANSON 1.4.1 is a free software released under the GNU GPL V2.

The permanent members of the VAUCANSON and Vgi Groups are:

Alexandre Duret-Lutz, LRDE, EPITA, Paris
Sylvain Lombardy, IGM, Université Paris-Est Marne-la-Vallée, Paris
Jacques Sakarovitch, LTCI, CNRS / Telecom-ParisTech, Paris
Hsu-Chun Yen, EE Dept., National Taiwan University, Taipeih

adl@lrde.epita.fr
lombardy@univ-mlv.fr
sakarovitch@enst.fr
yen@cc.ee.ntu.edu.tw
XMLCorrector: A Tool for Correcting XML Documents With Respect to a Schema

Joshua Amavi\textsuperscript{1} Béatrice Bouchou\textsuperscript{2} Agata Savary\textsuperscript{2}

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France\textsuperscript{1}
Université François Rabelais Tours, LI, Blois Campus, France\textsuperscript{2}
E-mail: joshua.amavi@univ-orleans.fr, \{beatrice.bouchou+agata.savary\}@univ-tours.fr

Abstract. XMLCorrector\textsuperscript{1} is an implementation of an algorithm allowing to correct an XML document with respect to schema constraints expressed as a DTD. Namely, given a well-formed XML document \(t\) seen as a tree, a schema \(S\) and a non negative threshold \(th\) the algorithm finds every tree \(t'\) valid with respect to \(S\) such that the edit distance between \(t\) and \(t'\) is no bigger than \(th\).

The algorithm is based on a recursive exploration of the finite-state automata representing structural constraints imposed by the schema, as well as on the construction of an edit distance matrix storing edit sequences leading to correction candidate trees. The algorithm is an extension of ideas announced in [1, 2]. It has been made public under the GNU LGPL v3 license. To the best of our knowledge, this is the first full-fledged study of the XML tree-to-schema correction problem.

Keywords: XML Processing; Tree-to-Schema Correction; Tree Edit Distance.

Operations on an XML tree and distance between two trees. The correction of an XML document \(t\) w.r.t. a set of schema constraints \(S\) consists in computing new documents that verify the set of structural specifications stated in \(S\) and that are close to \(t\). For correcting an XML tree, we allow three kind of elementary operations on nodes: (i) insertion of a node at a position in the tree, (ii) deletion of a leaf in the tree, (iii) relabeling of a node in the tree. A sequence of node operations transforms a tree \(t\) into another tree \(t'\). Figure 1 shows an example of an initial tree \(t\) and of a tree \(t'\) resulting from \(t\) by the application of an operation sequence. Each node operation has a cost. The cost of the node operation sequence is equal to the sum of the costs of each operation in the sequence. The distance between \(t\) and \(t'\) is defined as the minimal cost of all operation sequences allowing to transform \(t\) into \(t'\). For instance, if we admit cost 1 for each node operation, the operation sequence in Figure 1 has cost 3. Note that there exists no operation sequence transforming \(t\) into \(t'\) with a lower cost, thus the distance between \(t\) and \(t'\) is 3. The aim of XMLCorrector, for a given well-formed XML document, a DTD and a threshold, is to find all correction candidates, i.e. all valid documents whose distance from the original document does not exceed the threshold. The program provides the list of all operation sequences leading to such correction candidates, as well as the resulting XML documents themselves.

Applications. Applications of this problem are important and vary widely. We have: XML data integration, web service searching and composition, performing consistent queries on XML databases, XML document classification. But the most common scenario is inherently associated with the web: there is a constant need of evolution, for both XML documents and schemas. \%vspace0.2cm

Screenshots. (Figure 2)

References


\textsuperscript{1}http://www.info.univ-tours.fr/~savary/English/xmlcorrector.html
Fig. 1. Application of the node edit sequence $\text{nos} = \langle (\text{add}, 1, e), (\text{delete}, 2, /), (\text{relabel}, 0, d) \rangle$ on the tree $t$, $\text{cost}(\text{nos}) = 3$

Fig. 2. Main Window and Content of the Results tab