XSSR: an experimental system for string-rewriting – decision problems, algorithms, and implementation

D. Hofbauer, C. Kögl, K. Madlener, F. Otto, B. Reinert

June 4, 1999

1 Intended functionality

XSSR is an experimental system for string-rewriting. It is intended for computations with monoids and groups that are given in terms of finite presentations. It incorporates various algorithms for deciding properties of monoids, submonoids, and elements of monoids and groups that are based on rewriting methods. In [Cre97] a detailed description is given of the algorithms that are to be implemented in XSSR.

A monoid or a group is described internally as a collection of finite presentations. These presentations are either entered by the user, or they are generated from given presentations through the use of the operations provided by the system. These operations include the Tietze transformations and various Knuth-Bendix style procedures for completion, λ-completion and weak completion. Also some algebraic operations like free or direct products will be available to construct new monoids or groups from given ones. However, in order to ensure that the database of presentations is free from contradictions, only presentations generated by the system can be added to the collection of presentations describing a certain monoid. If the user wants to add another presentation of the same monoid or group, then this must be done in such a way that the system can verify that the new presentation does indeed present the same object. In particular, the system determines and stores the morphisms that realize the isomorphisms between the various presentations of the same monoid or group.

Investigating a specific monoid or group the user can ask questions about properties of elements, subsets, or the monoid or group as such. For example, he can ask whether the monoid considered is actually a group. If the information is already available, or if an algorithm for answering this question is known for one of the presentations listed for the monoid or group considered, the system will provide an answer immediately; otherwise, it will state that the answer is not yet known, and it will suggest ways of trying to find the answer. For example, if asked whether the monoid M given by a finite special presentation $(\Sigma; S)$ is a group, the system will use the algorithm for deciding the group property if the string-rewriting system S is λ-confluent; otherwise, it will state that the answer is unknown, but that the user may try to obtain a special λ-confluent presentation for M by running the λ-completion procedure, and that an answer will be obtained in case the completion procedure succeeds. In addition, if the monoid is indeed a group, the system will also determine a function that realizes the operation of forming inverses in this group for future use.

Accordingly, XSSR will consist of two main parts: a basic part managing the collections of presentations that also incorporates the various completion procedures, and an application part implementing the various algorithms that solve the decision problems under consideration and that use presentations of certain restricted forms. If a question is asked about a monoid or group given through a collection of presentations, then this collection will be searched for a presentation of the most appropriate form for answering this question. Observe that some decision problems can be solved for rather general classes of presentations, but that more efficient algorithms are available for more specialized classes of presentations. For example, this is the case for the group problem.

Accordingly, the different classes of presentations are organized as a hierarchy. However, since certain classes are incomparable under set inclusion, this hierarchy is not a linear one. For example, every special and λ-confluent string-rewriting system is monadic, but in general it will not be confluent. Thus, the class of all finite, special, and λ-confluent presentations is incomparable to the class of all finite, monadic, and confluent presentations. Naturally, the most general class is the class of all finite presentations, and below this we have the class of all finite presentations involving a noetherian and λ-confluent string-rewriting system. On the other hand, the most specialized class is the one of all finite, special, and confluent presentations describing groups.

Using XSSR it will be possible to run various rewrite-based algorithms described in the literature for non-trivial examples, thus obtaining data on their behaviour in practice. This information will help to devise strategies for improving these algorithms. Also XSSR will provide an opportunity to try out new ideas for attacking open problems. For example, one of the major open problems concerns the strategy of how to introduce new generators during the completion process. Some ideas are being discussed, but without a good experimental environment it seems to be impossible to judge their merits and their drawbacks.

There are various options for extending the functionality of XSSR. A presentation of a group G together with a finite set of strings in the generators of this presentation describes a subgroup H of G. Since H is a group in itself, one may be interested in obtaining a (group) presentation of H. For certain of the classes of presentations considered in XSSR, this is indeed possible. In fact, in some cases a presentation can be computed for H that is from the same syntactic class of presentations as the one for G [MaOt98]. Another option is the computation of Gröbner bases in monoid and group rings. The theory of Gröbner bases in non-commutative rings exploits ideas from the theory of string-rewriting to a large extent, so that XSSR can serve as an ideal basis for a system performing computations in structures of this form [MaRe98]. Finally, XSSR will provide interfaces to other systems, for example to the system HOMER, which computes homology groups of monoids that are given through finite convergent presentations [Buc91].

2 Design decisions

The design of a software system capable of providing the described kind of assistance to the monoid and group theory practitioner is a complex task for several reasons:

- The application area is reasonably well understood in terms of the theoretical underpinnings, but it is much less clear what the most common or typical 'business cases' are that have to be supported by the software. We envision a system that goes well beyond a simple collection of algorithms which the user may invoke. Therefore analysis of the way in which researchers in the application area actually work continues as the already stable parts of the system are being designed. This means that at this time the system requirements are very detailed for some aspects but much less so for others.

- It was clear right from the beginning of the project that the resulting software system would be rather large. In order not to fall prey to common project risks like lack of an architecture, difficult maintainability and extensibility, and low performance, we tried to apply recent software design techniques that facilitate the design and implementation of more robust, extensible, and maintainable systems. These techniques are now usually expressed as architecture patterns or design patterns [GHJV95], [BMRSS96].

- In order to be able to solve non-trivial problems in monoid and group theory the underlying algorithms should be implemented to work as efficiently (in terms of time but also space complexity) as possible. This requirement is usually in conflict with the wish for easy extensibility, for example the integration of future algorithmic innovations or variations. Knuth-Bendix-type completion procedures, which are needed as a crucial preprocessing step for virtually all of the algorithms of [Cre97], are the most prominent example in our system. There is a host of strategies available for actually implementing the completion procedure, and even for a fixed implementation there is usually a set of client-configurable parameters that influence the performance of a completion run profoundly. For the more difficult completion problems it is not clear which strategy and which parameter values will lead to the most rapid termination (if at all) of the computation. The user would also like the software to take advantage of additional knowledge that may speed up the completion process. Examples are the existence of inverse letters or the fact that we deal with a special string-rewriting system. We attempt to reconcile the seemingly conflicting requirements of high efficiency and easy extensibility by formulating a generic completion procedure (or a small set of such procedures) that provides several plug-in points with clearly documented interfaces that correspond to 'natural' subtasks of the completion procedure. Developers can then implement components that conform to the interfaces and easily integrate them into the system (the normalization facility or the facility for dealing with critical pairs are subtasks that come to mind).

3 Object-oriented design

For the design and the implementation we employ object-oriented techniques. This approach seems to be well-suited for computer algebra systems in general, and the abstractions of our application area readily map onto classes and objects. As implementation language we chose C++ because it supports the implementation of almost all of the object-oriented modeling techniques that are used today and (if employed with care and experience) allows the production of efficient code. C++ also provides powerful facilities for generic programming (in the form of class and function templates) that have been put to good use in our system, as far as e.g. reusability of components is concerned. There is one caveat, though: C++ is a very complex language and it takes dedication and experience (and the right literature!) to profit rather than to suffer from its use.

Currently, the areas that are most actively worked on are the various completion procedures; the component for handling a monoid's or group's collection of presentations and for selecting the 'best' algorithm for a client request (if there is more than one algorithm available); the automata and language package, which is central to most of the non-obvious algorithms and some of the specialized completion procedures; and the design of user interface components that are to be kept entirely separate from the application kernel.

We have already talked about the completion procedures, but we can add that it pays to take advantage of classical and also of more recent data structures 'on the market'. Without going into details we cite tries and other more sophisticated data structures for finite automata (useful for matching), suffix trees (useful for matching and various indexing operations), and various priority queue data structures (useful for organizing the set of waiting equations). We were forced to implement many of these data structures from scratch, since there are no reusable software components available for our kind of application. While doing so, we try to anticipate future needs for reusable components and strive to produce those rather than highly specialized solutions when this does not imply unreasonable performance losses. So far experiments show promising results in terms of execution speed, but we expect even better results in the near future (when suffix tree-based indices are used).

This discussion also applies to the automata and language package. To our dismay the freely available automata packages do neither match our functional requirements nor satisfy our need for clearly documented and readily usable plug-in components. But this is not surprising, as most of these packages were not designed with reusability in mind. So we are forced to design a package from scratch, but we hope to at least profit from the implementation experiences that the authors of the available packages have written down [MMPTV95, RaWo94, Wat97].

4 User interface

The user interface plays a central role when it comes to actually working with a software system. Existing computer algebra systems come in two flavours: either as a loose collection of standalone programs or with a command line interface.

Standalone programs have to be invoked from the operating system's command line by passing them command line parameters and configuration file names (which have to be created separately) for each and every computation (or batch thereof). This is the most common flavour for systems that are early in their development cycle. Many systems actually never leave this state in terms of usability, which is unfortunate. This way of organizing functionality is nevertheless appealing for the system developer because of its low implementation overhead. An advanced version would be the organization of the single application programs as a pipes-and-filters system in which the output of one program can directly be used as the input for the next one.

Another frequent way of presenting a computer algebra system's functionality to the user is some command line interpreter, not unlike an operating system's shell or a BASIC interpreter. This can be taxing for the user, however, as he is forced to learn yet another command language. In recent years, mainstream computer algebra systems have taken big steps forward in terms of user assistance, providing integrated help systems or application wizards guiding the user through some complicated sequence of steps, but progress is still to be made.

In our case, we have highly structured objects that we want to present to the user, namely collections of monoid or group presentations, and that the user would like to manipulate in a 'natural way' or at least without having to deal with system specific idiosyncrasies (like command language syntax). There are many available algorithms, but not all are applicable at all times; some algorithms may depend on many parameters or strategic choices that the system cannot make without user assistance. Considering this, we think that a user interface employing graphical elements (a GUI) will be the most effective way for the user to communicate with our system. Nevertheless, we should not commit the mistake to mix the application code with the user interface code as this would essentially prohibit the reusability of our core classes (like rule sets, completion processes etc.). There are proven software design techniques for separating application code from the user interface code (namely three-tier and, more generally, multi-tier design). In order to facilitate (or, put more conservatively, in order not to hamper) the rapid production of the user interface, the application kernel must provide software components for all the important application domain abstractions and workflows together with a clear documentation that are geared towards the needs of the user interface programmer. This separation of concerns then makes it possible to produce more than one kind of user interface for all or some aspects of our application (like a GUI for the casual user and a command interpreter with an integrated programming language for the 'power user'). At this time we are still evaluating the graphical user interface programming environments provided by Tcl/Tk [Tcl] and by the Qt Class Library [Qt], [Dal99].

References

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture.* Wiley, 1996.

[Buc91] M. Buchheit. *Algorithmen zur Berechnung von Invarianten für konvergente Wortersetzungs-Systeme.* Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, November 1991.

[Cre97] R. Cremanns. *Decision problems for string-rewriting systems.* Mathematische Schriften Kassel 10/97, Universität Kassel, September 1997. Available at: `http://www.db.informatik.uni-kassel.de/FG_TH/projekte/proj10.html`

[Dal99] M. K. Dalheimer. *Programming with Qt.* O'Reilly Verlag, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[MaOt98] K. Madlener and F. Otto. *Some Applications Of Prefix-Rewriting In Monoids, Groups, And Rings.* Reports On Computer Algebra, No. 22, Centre for Computer Algebra, University of Kaiserslautern, 1998.
Available at: `http://www.mathematik.uni-kl.de/~zca`.

[MaRe98] K. Madlener and R. Reinert. String rewriting and Gröbner bases – a general approach to monoid and group rings. In *Proceedings of the Workshop on Symbolic Rewriting Techniques, Monte Verita, 1995*, pages 127–180. Birkhäuser, 1998.

[MMPTV95] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the Program AMoRE. *Bericht Nr. 9507*, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, 1995.

[RaWo94] D. R. Raymond and D. Wood. Grail: A C++ Library for Automata and Expressions. *J. Symb. Computation*, 17:341–350, 1994.
See also: `http://www.csd.uwo.ca/research/grail/`

[Tcl] `http://www.tclconsortium.org/`

[Qt] `http://www.troll.no/products/qt.html`

[Wat97] B.W. Watson. FIRE Lite: FAs and REs in C++. In D. Raymond, D. Wood, and S. Yu, editors, *Proc. of the 1st Workshop on Implementing Automata, (London, Ontario, Canada, 29-31 August 1996)*, Lecture Notes in Computer Science Vol. 1260, Springer, 1997.
See also: `http://www.ribbitsoft.com/`