Implementation of Completion by Transition Rules + Control: 

**ORME**

Pierre LESCANNE  
*Centre de Recherche en Informatique de Nancy (CNRS)*  
*and*  
*INRIA-Lorraine*  
*Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy, France*  
email: lescanne@loria.crin.fr

In [Les89] it was shown how one can design completion procedures by using transition rules, a data structure and a tool box. The procedures are controlled by strategies expressed in a simple language. More specifically, I have designed four variants of standard completion procedures which are derived one from the other, following an empirical program derivation process; only three versions are presented in the cited paper, and the one I added since is a three line description of a straightforward and unefficient control I derived from a naive reading of the original paper of Knuth and Bendix which I use as a starting point in the derivation process. In addition, I gave two variants of unfailing completion procedures and I called the project ORME.

The goal of ORME was multiple. I wanted to show how a presentation by a set of inference rules is a nice and natural tool for designing completion procedures. The description of the procedure is indeed clear (although it is possible to do better as shown later in this paper), easy to modify and to understand. The readability of the code is especially important for a theorem prover since it measures the level of confidence the user may have in the reasoning and/or the computation it performs. In addition to the method, I wanted to offer tools for building software prototypes of rewrite based theorem provers, and actually RECOND [Ham90], a completion procedure and an environment for conditional rewrite systems, was built quickly using ORME. I wanted the tools that I am going to use in future experiments to be efficient and robust and I thought that a completion procedure would be a good test for pieces of software in such a framework. For instance, I have identified rather subtle and rare bugs in procedures I borrowed. My purpose was also didactic; the set of procedures and the methodology could indeed provide good support for a class on rewriting and completion. Last, I wanted software I can use to make experiments on strategies.

This last year, I wanted to go further in the ORME project and to provide tools for associative and commutative completion. In this paper, I would like to tell what I learned from this new experience and also to introduce the reader to the code of ORME in its AC version. The paper is as self contained as possible, for people who know about associative and commutative completion, but since it uses a lot of concepts presented in [Les89], it could be a good idea to read this paper first.
1 The ORME philosophy: an introduction

ORME started from the statement that since the description of completion procedures by transition rules is a good way for proving their completeness [Bac87], it should also be a good way for describing their implementation. For instance, I was once challenged to write a Gröbner basis algorithm and although I had no experience, it took me little time to be able to run some easy and classical examples [CSY89].

The main idea is therefore to define a data structure that represents the current state of the system at each time. A set of transition rules represent operations that are performed on this structure. They are the elementary components of a completion procedure. Therefore, one may view a completion as a machine that works on a data structure, with the transition rules as transitions of the machine. A control or a strategy (as it is usually called in the context of theorem provers) tells how the transition rules should be applied on the data structure in order to perform a completion. To improve a procedure one can change the strategy, but most of the time a big improvement requires changing the whole data structure, usually by breaking components into smaller ones. This division provides a tighter control by doing more elementary transitions. The transition rules themselves use a set of basic operations, the tool box. Building the tool box is a part of the activity of implementing a completion procedure, but life is easier when one reuses pieces built for other purposes; it is therefore wise to build the tools so that they can be reused by others.

This modular structure of the software made it very easily to master and entering the code is an easy task.

2 The data structure used in associative and commutative ORME

This paragraph gives a description of the data structure used in ORME for associative and commutative completion. It is based on the ANS-completion, a standard completion that gives a high priority to simplification (see [Les89]). The main difference is that each component of the data structure has an extension which contains extended rules as required by the theory of associative and commutative completion. With respect to the previous versions of ORME, I have made a major improvement by using the CAML data type labeled product instead of the plain cartesian product (generalized pairs). The code is therefore more readable because the components are more clearly identified. This also allows taking the freedom of dynamic local changes in the structure, making the program more efficient. The components that can be dynamically locally changed are called mutable. The data structure in ORME for associative and commutative completion is described in figure 1 and its components correspond to the following concepts.

- **Theory** is a list of operators that are associated with an associative and commutative operation.
- **E** is a set of identities. They are either given by the user as an input or produced by the completion procedures as critical pairs.
- **S** and **S_ext** contain simplifiers, which are rules that just come from **E** after orientation and which are used to simplify the data structure. **S** contains at most one rule and **S_ext** contains at most a pair made of a rule (say \( s \rightarrow t \)) and its extension (say