# The Third Rewrite Engines Competition

Francisco Durán[1], Manuel Roldán[1], Jean-Christophe Bach[2], Emilie Balland[2], Mark van den Brand[3], James R. Cordy[4], Steven Eker[5], Luc Engelen[3], Maartje de Jonge[7], Karl Trygve Kalleberg[6], Lennart C. L. Kats[7], Pierre-Etienne Moreau[2], and Eelco Visser[7]

[1] LCC, Universidad de Málaga, Málaga, Spain
[2] INRIA Nancy - Grand Est, Villers-lès-Nancy Cedex, France
[3] Dpt. of Mathematics and Comp. Sci., Eindhoven U. of Technology, The Netherlands
[4] School of Computing, Queen's University at Kingston, Canada
[5] Computer Science Laboratory, SRI International, Menlo Park, CA, USA
[6] University of Bergen, Bergen, Norway
[7] Dpt. of Software Technology, Delft University of Technology, The Netherlands

**Abstract.** This paper presents the main results and conclusions of the Third Rewrite Engines Competition (REC III). This edition of the competition took place as part of the 8th Workshop on Rewriting Logic and its Applications (WRLA 2010), and the systems ASF+SDF, Maude, Stratego/XT, Tom, and TXL participated in it.

## 1 Introduction

As in the 2006 and 2008 editions of the Workshop on Rewriting Logic and its Applications [9, 13], in WRLA 2010 a rewrite engines competition was organized, with the aim of bringing to the community the different rewrite engines available, with the main purpose of showing the strengths of each of the participating systems. And as in WRLA 2006 and WRLA 2008, the 2010 edition of the workshop included a session on the competition, in which, in addition to a presentation on the organization, development, and results of the competition, the developers of each of the systems in it had the opportunity of presenting their systems. The discussion and questions from the audience where without any doubt the most interesting part of the session. The present paper tries to summarize such a competition and session, providing additional details on the way the competition was organized and conducted, and trying to complete on the discussion and comparison of the different systems and the results obtained.

The Third Rewrite Engines Competition counted with the particiption of five systems, namely ASF+SDF [20, 19], represented by M. van den Brand and L. Engelen; Maude [4, 5], represented by F. Durán and S. Eker; Stratego/XT [22, 2], represented by M. de Jonge, K. T. Kalleberg, L. Kats, and E. Visser; Tom [1], represented by J.-C. Bach, E. Balland, and P.-E. Moreau; and TXL [7, 6], represented by J. Cordy. The second edition gathered the same number of participants (ASF+SDF, Maude, Stratego, TermWare [14] and Tom) and two in the first one (ASF+SDF and Maude). We would have liked to gather more systems,

although it is not easy, for different reasons. We would like to thank the developers of Kiama [15], Rascal [12] and TermWare, who showed their interest in being involved, but for one reason or another, were not able to get to the end. Developers of other systems were also invited, but kindly refused their participation. Our apologies to any other system that should have been invited but was not... perhaps in the next one!

This edition of the competition, as the previous ones, was very illustrative, since it showed that each of the engines focusses on very specific problems, and that they are very good at them. More than a competition, REC is an opportunity to show the different systems and their strengths, and why not, their weaknesses, to the rewriting community.

The first competition [10], which was organized by G. Roşu, focused on efficiency, specifically speed, memory management and built-ins use. There were only two participants, ASF+SDF, represented by M. van den Brand, and Maude, represented by S. Eker, but awoke interest on such a kind of event and opened the door to the subsequent competitions. For this first edition of the competition, a number of test examples were compiled, all of them using features supported by both systems. Most of the problems used came from the benchmarks of the two systems. The paper [10] includes very interesting discussions on the technical details why Maude and ASF+SDF behaved like they did on the different tests run in the 1st REC. Since many of the problems used in it are again in this 3rd REC, the discussions there are a very useful complement to the present paper.

For the second edition [11], the possibility of having some bigger problems to develop was considered. Several ideas were considered, as the development of a small theorem prover, the exploration of a search space, a transformation of XML or a tree... Among all these problems, first steps in the world of program transformations were taken. In the end, a common language to specify term rewrite systems, called REC, was developed, and the development of an interpreter for REC was proposed. Then, the set of rewrite problems proposed was expressed in this REC language.

The REC language and its use to run the problems in the competition was maintained on this 3rd competition. Some new problems were included in our benchmark, but basically the efficiency of the systems was compared running the different problems in this REC syntax on the interpreters developed for REC in the participating systems. To be able to reuse the interpreters developed for the 2008 competition, the syntax of the REC language, which is described in Section 3.1, was not modified.

After the experience with the REC language, and since some of the systems in the competition specialize on program transformation, we decided to include in this edition some additional problems related to the definition of programming languages, and to the generation, analysis and transformation of programs, which is one of the key application areas of term rewriting. Following a suggestion by J. Cordy, we decided to include some problems from the TIL Chairmarks, developed by J. Cordy and E. Visser. The TIL language and the TIL Chairmarks problems used in the competition are described in Sections 3.2 and 5.

## 2 The Systems in the Competition

The systems in the 3rd REC are of a very different nature. We have compilers and interpreters, we have specific-purpose and general-purpose systems, we have embedded rewriting systems and stand-alone systems, ... The results here should not be taken as a final comparison of the systems, but just as a starting point on some very specific issues. In fact, there are many strong points in each of the systems that are not considered in the competition. For example, SDF+SDF, Stratego, Tom, and TXL have very sophisticated facilities for program manipulation, with, e.g., very powerful parsers and pretty-printing tools; Tom is embedded into different generalist programming languages (e.g. C, Java, Python, C++, C#); Maude supports matching modulo any combination of associativity, commutativity, and identity, and unification modulo commutativity and associativity-commutativity, and provides a suite of formal tools. In this section we introduce the main features of each of the systems.

### 2.1 ASF+SDF

ASF+SDF is a general-purpose, executable, algebraic specification formalism based on (conditional) term rewriting. Its main application areas are the definition of the syntax and the static semantics of (programming) languages, program transformations and analysis, and for defining translations between languages.

The ASF+SDF formalism [21] is a combination of two formalisms: ASF (the Algebraic Specification Formalism) and SDF (the Syntax Definition Formalism). SDF is used to define the concrete syntax of a language, whereas ASF is used to define conditional rewrite rules; the combination ASF+SDF allows the syntax defined in the SDF part of a specification to be used in the ASF part, thus supporting the use of user-defined syntax when writing ASF equations. ASF+SDF also supports modular structuring of specifications using names modules, and thus enabling reuse.

The ASF+SDF and the ASF+SDF Meta-Environment have been applied in a broad range of applications. The application areas can be characterized as: prototyping of domain specific languages, software renovation, and code generation. An overview of some of the applications is given in [17]. The ASF+SDF system, its documentation, and related papers are available at http://www.meta-environment.org/. ASF+SDF is no longer maintained and is replaced by Rascal, see http://www.rascal-mpl.org/.

### 2.2 Maude

Maude is a language and a system based on rewriting logic [4, 5, 3]. Maude modules are rewrite theories, while computation with such modules corresponds to efficient deduction by rewriting. Since rewriting logic contains equational logic, Maude also supports equational specification and programming in its sublanguage of functional modules and theories. The underlying equational logic of

Maude is membership equational logic, that has sorts, subsorts, operator overloading, and partiality definable by membership and equality conditions. Because of its logical basis and its initial model semantics, a Maude module defines a precise mathematical model. This means that Maude and its formal tool environment can be used in three, mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification system. The Maude system, its documentation, and related papers and applications are available from the Maude website http://maude.cs.uiuc.edu.

Maude provides very efficient support for rewriting modulo any combination of associativity, commutativity, and identity axioms, and provides two built-in rewrite strategies: top-down rule fair and position fair. Maude's rewrite engine makes extensive use of advanced semi-compilation techniques and sophisticated data structures supporting rewriting modulo. Besides supporting efficient execution, Maude also provides a range of formal tools and algorithms to analyze rewrite theories and verify their properties including a search facility for doing breadth first search with cycle detection, and a linear time temporal logic model checker.

### 2.3 Stratego/XT

Stratego/XT is a language and toolset for program transformation. The Stratego language provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations, thus supporting the development of transformation components at a high level of abstraction.

The XT toolset offers a collection of extensible, reusable transformation tools, such as powerful parser and pretty-printer generators and grammar engineering tools. Stratego/XT supports the development of program transformation infrastructure, domain-specific languages, compilers, program generators, and a wide range of meta-programming tasks.

Stratego has two backends: one for generating C code (StrC), and another for generating Java code (StrJ). The Stratego/XT system, its documentation, and related papers are available at http://strategoxt.org/.

### 2.4 Tom

Tom [1] is an extension of Java which adds support for algebraic data-types and pattern matching. Contrary to other languages, Tom does not enforce any particular tree representation for the objects being matched. To make this possible, Tom provides a mapping definition formalism to describe the relationship between the concrete Java implementation and the algebraic view, which allows to define transformations directly on existing Java data-structures. The other features of the Tom language are mainly a powerful pattern-matching construct

(matching modulo theory, list-matching, anti-patterns, XML notation,...); support for private types in Java; an efficient implementation of typed and maximally shared terms, an extension for term-graph rewriting and a strategy language inspired by Elan and Stratego.

To conclude, the main originality of Tom is that it is piggybacked on top of Java, which allows to integrate smoothly declarative transformation code in existing Java programs. It has been used to implement many large and complex applications, among them the compiler itself. Tom is used in academic projects to prototype models based on rewriting but it is also successfully integrated in industrial products (for example, database request translation in SAP's software). The Tom systems is available at http://tom.loria.fr/.

### 2.5 TXL

TXL [6, 7] is a special-purpose programming language designed for creating, manipulating and rapidly prototyping language descriptions, tools and applications using source transformation. TXL is designed to allow explicit programmer control over the interpretation, application, order and backtracking of both parsing and rewriting rules. Using first order functional programming at the higher level and term rewriting at the lower level, TXL provides for flexible programming of traversals, guards, scope of application and parameterized context. This flexibility has allowed TXL users to express and experiment with both new ideas in parsing, such as robust, island and agile parsing, and new paradigms in rewriting, such as XML markup, rewriting strategies and contextualized rules, without any change to TXL itself. TXL's website is http://txl.ca.

## 3 The REC and TIL languages

With different goals in mind, two different languages, REC and TIL, have been used in the competition. We present these simple languages in the following sections. Section 3.3 discusses the lexical analysis and parsing tools developed for these languages as part of the competition.

### 3.1 The REC language

REC is a term rewriting language, that was defined for the second rewrite engines competition as a common language in which to write the rewrite tasks to pose to the participant systems. The REC language is many-sorted, does not have any built-ins, uses prefix syntax, does not support overloading, allows conditional rules, and includes syntax for `assoc`, `comm`, `id`, and `strat` attributes *à la* OBJ. A BNF description of the syntax of the language is given in Figure 1. Figure 2 shows the REC specification of the factorial function, with the natural numbers, with plus and times operations, represented using Peano notation.

Each of the participants was asked to build a program transforming the problems in this REC syntax to the language of their corresponding tools. Those

```
⟨spec⟩          ::= REC-SPEC ⟨id⟩
                      [ SORTS ⟨idlist⟩ ]
                      [ VARS ⟨vardecllist⟩ ]
                      [ OPS ⟨opdecllist⟩ ]
                      [ RULES ⟨rulelist⟩ ]
                    END-SPEC
⟨idlist⟩        ::= ⟨id⟩ ⟨idlist⟩ | ϵ
⟨vardecllist⟩   ::= ⟨idlist⟩ : ⟨id⟩ ⟨vardecllist⟩ | ϵ
⟨opdecllist⟩    ::= ⟨opdecl⟩ ⟨opdecllist⟩ | ϵ
⟨opdecl⟩        ::= op ⟨id⟩ : ⟨idlist⟩ -> ⟨id⟩
                    | op ⟨id⟩ : ⟨idlist⟩ -> ⟨id⟩  ⟨opattrlist⟩
⟨opattrlist⟩    ::= ⟨opattr⟩ ⟨opattrlist⟩ | ϵ
⟨opattr⟩        ::= assoc | comm | id( ⟨term⟩ ) | strat( ⟨intlist⟩ )
⟨rulelist⟩      ::= ⟨rule⟩ ⟨ruleslist⟩ | ϵ
⟨rule⟩          ::= ⟨term⟩ -> ⟨term⟩ | ⟨term⟩ -> ⟨term⟩ if ⟨condlist⟩
⟨condlist⟩      ::= ⟨cond⟩ | ⟨cond⟩ , ⟨condlist⟩
⟨cond⟩          ::= ⟨term⟩ -><- ⟨term⟩                          % ==
                    | ⟨term⟩ ->/<- ⟨term⟩                       % =/=
⟨term⟩          ::= ⟨id⟩ | ⟨id⟩ ( ) | ⟨id⟩ ( ⟨termlist⟩ )
⟨termlist⟩      ::= ⟨term⟩ | ⟨term⟩ , ⟨termlist⟩
⟨intlist⟩       ::= ⟨int⟩ ⟨intlist⟩ | ϵ
⟨command⟩       ::= get normal form for: ⟨term⟩
                    | check the confluence of: ⟨term⟩ -><- ⟨term⟩
```

⟨*id*⟩ are non-empty sequences of any characters except ' ', '`(`', '`)`', '{', '}', '"' and '`,`'; and excluding '`:`', '`->`', '`-><-`', '`->/<-`', '`if`', and keywords `REC-SPEC`, `SORTS`, `VARS`, `OPS`, `RULES`, and `END-SPEC`.
⟨*int*⟩ are non-empty sequences of digits.
Comments are given using '`%`'. Text written in the line after a '`%`' is discarded.

**Fig. 1.** BNF description of the syntax of the REC language.

that already developed this program transformer for REC II were able to use the same tool, since the syntax of the language did not change. This was one of the reasons for developing such a language in 2008. ASF+SDF and TXL had to build it from scratch for REC III.

### 3.2 TIL

The Tiny Imperative Language (TIL) is a very small imperative language with assignments, conditionals, and loops, designed by J. Cordy and E. Visser, as a basis for small illustrative example transformations. These example transformations define the benchmark transformation tasks they propose as the TIL Chairmarks. As we will explain in Section 5, a selection of the TIL Chairmarks has been used in this 3rd REC. The syntax of TIL is given in Figure 3. A some more detailed description of the language is available at http://www.program-transformation.org/Sts/TinyImperativeLanguage.

```
REC-SPEC Factorial
SORTS Nat
OPS
  0 : -> Nat              % zero
  s : Nat -> Nat          % succesor
  plus : Nat Nat -> Nat   % addition
  times : Nat Nat -> Nat  % product
  fact : Nat -> Nat       % factorial
VARS N M : Nat
RULES
  plus(0, N) -> N
  plus(s(N), M) -> s(plus(N, M))
  times(0, N) -> 0
  times(s(N), M) -> plus(M, times(N, M))
  fact(0) -> s(0)
  fact(s(N)) -> times(s(N), fact(N))
END-SPEC
```

**Fig. 2.** REC specification of the factorial function.

### 3.3 Lexical analysis and parsing

We had two different approaches in the competition for the implementation of the translators requested for REC and TIL. While ASF+SDF, Stratego/XT, Tom, and TXL representatives built programs that transformed the original programs and commands, and were later loaded and executed, in Maude a programming environment was built, able to read REC programs and commands and give outputs. Maude does not have facilities to handle files, what complicates the reading of input files and the generation of output files with the resulting programs. However, Maude has some facilities for building execution environments, that was the approach followed in that case.

Maude has some limitations at the lexical level, what forced the Maude representatives to alter the input files (enclosing the input programs in parentheses and removing comments). Maude and ASF+SDF does not offer constructs to read input from the command line while rewriting, which makes it impossible to implement the interpreter for TIL as it is implemented in Tom or TXL. Alternatively, in the Maude and ASF+SDF cases, interpreters that take a program and a list of values as input, and provide the output for that program given the input as its result, were implemented.

No lexical or parsing problems were encountered in the cases of ASF+SDF, Stratego/XT, Tom, and TXL. ASF+SDF and Stratego/XT are based on SDF and SGLR,[8] and support the full class of context-free grammars. Tom uses the

[8] SGLR (Scannerless Generalized LR Parser) is an implementation of the Generalized LR algorithm [16] with extensions for scannerless parsing.

```
⟨program⟩ ::= ⟨statement_list⟩
⟨statement_list⟩ ::= ⟨statement⟩ ⟨statement_list⟩ | ϵ
⟨statement⟩ ::= ⟨declaration⟩ | ⟨assignment_statement⟩ | ⟨if_statement⟩
              | ⟨while_statement⟩ | ⟨for_statement⟩ | ⟨read_statement⟩
              | ⟨write_statement⟩
⟨declaration⟩ ::= var ⟨identifier⟩ ;              % Untyped variables
⟨assignment_statement⟩ ::= ⟨identifier⟩ := ⟨expression⟩ ;
⟨if_statement⟩ ::= if ⟨expression⟩ then ⟨statement_list⟩ end
                 | if ⟨expression⟩ then ⟨statement_list⟩
                   else ⟨statement_list⟩ end
⟨while_statement⟩ ::= while ⟨expression⟩ do ⟨statement_list⟩ end
⟨for_statement⟩ ::= for ⟨identifier⟩ := ⟨expression⟩ to ⟨expression⟩ do
                        ⟨statement_list⟩
                    end
⟨read_statement⟩ ::= read ⟨identifier⟩ ;
⟨write_statement⟩ ::= write ⟨expression⟩ ;
⟨expression⟩ ::= ⟨primary⟩ | ⟨expression⟩ ⟨op⟩ ⟨expression⟩
⟨primary⟩ ::= ⟨identifier⟩ |  ⟨integer⟩ | ⟨string⟩ | ( ⟨expression⟩ )
⟨op⟩ ::=   = | != | + | - | * | /   % from lowest to highest priority
```

**Fig. 3.** Grammar for Tiny Imperative Language (TIL).

ANTLR parser generator;[9] the abstract syntax tree (AST) produced by ANTLR can be directly reused in the Tom system. TXL has its own top-down programmable parser that the user can control directly [8] as part of the TXL program.

## 4 The REC problems

The REC language presented in Section 3.1 has been used in two different ways in this 3rd rewrite engines competition. First, the participants were asked to write interpreters for it, so that REC can be used as a common language in which to write the problems used to compare their performance. Since all interpreters for all the systems were provided, there was no need for hand-made transformations. In the 2008 competition some of the systems did not develop such interpreters, and solutions were provided by hand; the rest of the systems were allowed to provide optimizations of the automatically generated rewrite systems. In the 2006 competition all the specifications were written by hand in each of the participating systems.

Translating the REC specifications to their counterparts in the different systems is an easy task, and the automatic translations take little time. The implementations of these translations are quite straightforward in all the systems,

[9] The web site of ANTLR (ANother Tool for Language Recognition) is at http://www.antlr.org/.

and optimization was not attempted in any case. In all the cases, all terms are represented by their concrete syntax all the time. E.g., natural numbers are represented using Peano notation. Manual optimizations using built-ins, memoization, etc. could have been considered for all the systems but were not.

### 4.1 Disclaimer

We must acknowledge that there was perhaps somewhat of a mismatch between the REC test cases and the normal applications of the Stratego and TXL systems. These systems are not traditional rewrite engines, and are typically not applied for traditional rewriting problems but for other applications such as program transformation and analysis. Our test set in this section focuses purely on raw rewriting power, and as such may be biased towards traditional rewriting systems.

For Stratego, an *innermost* strategy is used to emulate the behavior of a true term rewriting engine. Likewise, for TXL, term rewriting is implemented using a global transformation rule that globally applies the entire ruleset to a fixed point. REC rewrite rules are directly mapped to rules in the different systems, but in the case of Stratego and TXL, the individual rules are combined using functional composition. Although the order of application can affect performance, no attempt has been made to optimize this order in the mechanical translation from REC. Stratego and TXL do not apply memoization when evaluating the rules.

Maximal sharing of identical subterms ensures efficient memory usage and constant time comparison at the cost of slightly increased time spent when constructing new terms. Since the tests in our benchmark involve large terms with repeating subterms and do not use line numbers or other context information, systems that employ maximal sharing may be at the advantage. Stratego (when compiled to C) and ASF+SDF implement maximal sharing based on the ATerm library [18]. TXL and the Java version of Stratego do not employ maximal sharing. Tom provides an efficient implementation of typed and maximally shared terms in Java.

As in REC II, the rewriting problems are organized in four categories: unconditional rewriting (TRS), conditional rewriting (CTRS), rewriting modulo (Modulo), and context-sensitive rewriting/rewriting with local strategies (CS). Only Maude has support for the features needed to be in all these categories. ASF+SDF, Stratego and TXL only participate in the TRS and CTRS categories. Tom supports rewriting modulo associativity since its first version. In a recent release it also provides support for rewriting modulo associativity-commutativity. However, although the implementation is correct, it is not yet very efficient.

### 4.2 Results for the rewriting problems

We now present the results for each of the rewrite examples considered in the competition. Although we have five participants, namely ASF+SDF, Maude, Stratego, Tom and TXL, two different versions were considered for both Maude

and Stratego. In the case of Maude we used 32-bits and 64-bits binaries, and for Stratego we tested a C implementation and a newly developed Java version.

The five systems were installed on a 64-bits Linux 2.40GHz/4GB Intel Core 2 Quad. The installation of the systems was done by M. Roldán, who also ran most of the tests.

For each case, after a brief description of the problem, a table with the times used in the computations is presented. In these tables, all times are given in milliseconds. Those test cases that either took long (more than one hour), ran out of memory, or produced an internal error show as '—'.

In most of these cases, a manual implementation in the system's language, rather than a (naive) automatic translation of the REC specification, would be more appropriate. In some cases we may get huge improvements by reordering the equations, saving partial computations, using memoization, etc. In the 2nd REC we consider both an automatic translation and a handwritten optimized version for each of the problems in the competition. In this edition we are only considering the automatic translation. See [10] for the results and comparison in the 2008 edition of the competition.

In most cases, the numbers are self explanatory. In some of them we give some explanations or provide some pointers for a discussion on them. We present a selection of the results in this paper, and refer to the web site of the competition, at http://www.lcc.uma.es/rewriting_competition, for further details. All the files and results of the competition are available in this web site, where one can find a table that includes, for each of the problems, the specification and the tests run on it in REC syntax, and the corresponding problems in the syntax of each the participant systems, together with the times consumed in their computation and the solutions given.

**TRS: unconditional rewriting.** In this category we have rewrite systems for the calculation of the factorial of a natural number, the n-th number in the Fibonacci sequence, a function reversing a list, an artificial rewrite system to test garbage collection algorithms, and an ASF+SDF benchmark for the study of resource usage in brute-force rewriting (no built-ins, no strategies).

*Factorial.* The specification of the factorial of a natural number was presented in Figure 2. The factorial function is calculated for values 6, 8, 10, and 12.

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 6 | 17 | 0 | 0 | 0 | 20 | 5 | 4,566 |
| 8 | 26 | 4 | 5 | 50 | 170 | — | — |
| 10 | 32,466 | 544 | 754 | — | — | — | — |
| 12 | — | — | — | — | — | — | — |

The reason why the Maude interpreter outperforms the ASF+SDF and Tom compilers is probably because of the term representation they used. See [10] for a more in depth discussion on this case for the ASF+SDF and Maude systems.

*Fibonacci.* The Fibonacci sequence is specified by the following three rules:

```
fibb(0) -> s(0)
fibb(s(0)) -> s(0)
fibb(s(s(N))) -> plus(fibb(s(N)), fibb(N))
```

The `fibb` function is calculated for values 10, 20, 30, 40, and 50.

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 10 | 10 | 0 | 0 | 0 | 20 | 2 | 7 |
| 20 | 86 | 10 | 7 | 20 | 90 | — | 108,196 |
| 30 | 10,788 | 2,273 | 2,505 | — | — | — | — |
| 40 | — | — | — | — | — | — | — |

*Garbage collection.* This rewrite system consists of the following rules:

```
c(0, Y) -> Y
c(s(X), Y) -> s(c(X,Y))
f(X, Y, Z, T, U) -> f(X, Y, Z, Y, Z, T, U)
f(X, Y, s(Z), N, P, T, U) -> f(X, Y, Z, N, P, c(T, T), U)
f(X, s(Y), 0, N, P, T, U) -> f(X, Y, P, N, P, T, T)
f(s(X), 0, 0, N, P, T, U) -> f(X, N, P, N, P, 1, 0)
f(0, 0, 0, N, P, T, U) -> T
```

The different tests run consist in the reduction of terms of the form `f(m,n,p,0,1)`, with different values for `m`, `n`, and `p`.

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| f(2,2,2,0,1) | 14 | 0 | 0 | 0 | 10 | 17 | 7 |
| f(2,2,4,0,1) | 39 | 1 | 1 | 0 | 0 | 50 | 13,378 |
| f(2,4,2,0,1) | 20 | 0 | 1 | — | — | 26 | 661 |
| f(2,4,4,0,1) | 9,019 | 261 | 300 | — | — | — | — |
| f(4,2,2,0,1) | 15 | 0 | 0 | — | — | 18 | 8 |
| f(4,2,4,0,1) | 44 | 2 | 1 | — | — | 57 | 14,495 |
| f(4,4,2,0,1) | 16 | 1 | 0 | — | — | 26 | 727 |
| f(4,4,4,0,1) | 8,918 | 459 | 512 | — | — | — | — |

Notice that all the systems behave quite well for all the tests except for those with $\mathtt{n} = 4$ and $\mathtt{p} = 4$.

*List reverse.* Given lists represented with constructors `cons : Nat List -> List` and `nil : -> List`, the following `rev` function reverses the elements of a list of natural numbers.

```
conc(cons(E, L), L') -> cons(E, conc(L, L'))
conc(nil, L') -> L'
reverse(cons(E, L)) -> conc(reverse(L), cons(E, nil))
reverse(nil) -> nil
```

The tests are run on lists of $10^2$, $10^3$, and $10^4$ elements.

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| $10^2$ | 23 | 0 | 0 | 0 | 10 | 39 | 1,495 |
| $10^3$ | 780 | 46 | 31 | — | — | 622 | — |
| $10^4$ | 69,403 | 4,520 | 3,714 | — | — | 105,930 | — |

*ASF+SDF benchmark for brute force rewriting.* In these tests we include three different functions: symbolic evaluation of $2^n$ modulo 17 (sym), for testing speed of rewriting with almost no memory usage; symbolic evaluation of $2^n$ modulo 17 after expanding the expression (eval), to test memory management; and computation on huge $2^n$, not-alike trees (tree), also to test memory management. The specification of these problems can be found in [19]. An interesting discussion on the behavior of ASF+SDF and Maude on these tests can be found in [10].

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| sym(10) | 22 | 3 | 3 | 0 | 10 | 90 | 4,714 |
| sym(20) | 337 | 3,162 | 2,506 | 1,830 | 4,480 | 5,877 | — |
| eval(10) | 16 | — | — | 0 | 80 | 84 | — |
| eval(20) | 346 | — | — | 2,190 | 10,210 | 4,488 | — |
| tree(10) | 56,734 | 5 | 5 | — | — | 113 | — |
| tree(20) | — | 9,674 | 12,480 | — | — | 5,818 | — |

ASF+SDF performs much better than the others for `sym` and `eval`. However, rewriting `tree(10)` and `tree(20)` take a lot of time in all the systems using the automatically generated specifications because many computations are repeated. It is remarkable that Tom and Maude perform better than ASF+SDF in these tests. Saving the computations to avoid the repetition of the evaluations would result in big improvements for all the systems. E.g., just by introducing variables that store the rewritten result of such subterms `tree(20)` takes 15 milliseconds in ASF+SDF.

**CTRS: conditional term rewrite systems.** In this category we find bubblesort, mergesort, quicksort, a bit matrix closure algorithm, an odd/even artificial problem, and a specification of the towers of Hanoi problem.

*Bubblesort.* Given lists of natural numbers defined by `cons` and `nil` as above, and given a less-than function `lt`, the bubblesort algorithm is specified by the single following rule:

```
cons(N, cons(M, L)) -> cons(M, cons(N, L)) if lt(M, N) -><- true
```

The following results are obtained for lists of 10, 100, and 1,000 elements in reverse order:

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 10 | 13 | 0 | 0 | 0 | 50 | 35 | 19 |
| 100 | 26 | 85 | 74 | 110 | 500 | 88 | — |
| 1,000 | 1,550 | 383,815 | 450,887 | 130 | 330 | 5,299 | — |

Maude performs so badly in this case because of the very ineffective way in which it treats conditional rules.

*Mergesort.* Given lists of natural numbers defined by `cons` and `nil` as above, and a less-than-or-equal predicate on natural numbers `lte`, the `mergesort` function is specified as follows:

```
merge(nil, L) -> L
merge(L, nil) -> L
merge(cons(X, L1), cons(Y, L2)) -> cons(X, merge(L1, cons(Y, L2)))
  if lte(X, Y) ->< true
merge(cons(X, L1), cons(Y, L2)) -> cons(Y, merge(cons(X, L1), L2))
  if lte(X, Y) ->< false
split(cons(X, cons(Y, L)))
  -> pair(cons(X, p1(split(L))), cons(Y, p2(split(L))))
split(nil) -> pair(nil, nil)
split(cons(X, nil)) -> pair(cons(X, nil), nil)
mergesort(nil) -> nil
mergesort(cons(X, nil)) -> cons(X, nil)
mergesort(cons(X, cons(Y, L)))
  -> merge(mergesort(cons(X, p1(split(L)))),
           mergesort(cons(Y, p2(split(L)))))
p1(pair(L1, L2)) -> L1
p2(pair(L1, L2)) -> L2
```

The following results are obtained for lists of 10, 100, and 1,000 elements in reverse order:

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 10 | 9 | 1 | 0 | 0 | 70 | 50 | 8 |
| 100 | — | 9 | 9 | — | — | — | — |
| 1,000 | — | 9,134 | 10,721 | — | — | — | — |

The reason why most of the systems perform so badly is because the equations for `split` and `merge` are not right-linear. The rewriting of the `split(L)` terms is repeated if the sharing is not detected as in Maude. Simple modifications in the specifications, using memoization or intermediate variables, would lead to big improvements. E.g., in ASF+SDF, the use of these variables takes the computation times to 11/7/20.

*Quicksort.* The following results are obtained for lists of 10, 100, and 1,000 elements in reverse order:

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 10 | 10 | 0 | 1 | 0 | 230 | — | 305 |
| 100 | — | 42 | 39 | — | — | — | — |
| 1,000 | — | 193,616 | 227,166 | — | — | — | — |

As for the mergesort function above, the reason for such results is that many computations are repeated many times. By introducing new variables to avoid re-computations in, e.g., ASF+SDF makes the times to go down to 15/19/32.

*Bit matrix closure.* This rewrite system calculates the reflective and transitive closure of a bits matrix. The results for sizes 10x10, 20x20 and 30x30 are:

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 10x10 | 19 | 2 | 1 | — | — | 28 | 1,504 |
| 20x20 | 32 | 12 | 10 | — | — | 84 | 56,907 |
| 30x30 | 8 | 59 | 43 | — | — | 103 | 809,494 |

*Odd/even.* This is an artificial example to test the exponential explosion that can result due to conditional rewriting.

```
odd(0) -> false
even(0) -> true
odd(s(N)) -> true if even(N) -><- true
even(s(N)) -> true if odd(N) -><- true
odd(s(N)) -> false if even(N) -><- false
even(s(N)) -> false if odd(N) -><- false
```

The results obtained are the following:

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| odd(15) | 13 | 69 | 60 | 0 | 10 | 15 | 0 |
| odd(20) | 11 | 0 | 0 | 0 | 0 | 14 | 5,880 |
| odd(25) | 14 | 66,828 | 53,864 | 0 | 0 | 18 | 0 |

ASF+SDF and Tom do well in this example because they optimize the compiled code to avoid re-computation in conditions. Maude rewrites the computations as given. In the case of TXL, evaluating the `odd` function on an odd number results in an almost immediate success, while evaluating it on an even number results in an exponential search. In the case of Maude it is the other way around, because the rules are considered is a different order.[10] The use of memoization, or simply changing the order of the rules, significantly improves the efficiency of Maude in this case.

[10] The program transformation implemented for Maude uses a set of rules instead of a list, and in cases like this it may change the order in which the rules are considered.

*Hanoi towers.* This rewrite system solves the traditional problem of the towers of Hanoi. The solutions were executed for 4 and 16 disks.

| | ASF+SDF | Maude32 | Maude64 | StrC | StrJ | Tom | TXL |
|---|---|---|---|---|---|---|---|
| 4 | 8 | 0 | 0 | 0 | 20 | 26 | 6 |
| 16 | 950 | 188 | 210 | — | — | 378 | — |

**Modulo: rewriting modulo associativity and/or commutativity and/or identity.** Maude and Tom are the only systems between the participants providing some form of rewriting modulo. Maude supports rewriting modulo any combination of associativity, commutativity and identity. Tom supports rewriting modulo associativity, and a first attempt for rewriting modulo associativity-commutativity in its latest release.

*Tautology-hard, darts, 3-value logic, and permutations.* The tautology-hard rewrite system evaluates Boolean expressions with associative and commutative `and`, `xor`, `or`, and `iff` operations. Logic3 defines a 3-value logic, and darts operates on sets. Their specifications include several associative and commutative operators. The permutations specification defines a function that calculates all the permutations of a list. It uses two operators which are declared associative and with identity element. The tautology-hard rewrite system is evaluated on three expressions of different sizes. These are all the results obtained:

| | Maude32 | Maude64 | Tom |
|---|---|---|---|
| tautology-hard 1 | 10 | 9 | 451 |
| tautology-hard 2 | 200 | 173 | — |
| tautology-hard 3 | 523 | 471 | — |
| darts | 2 | 2 | 56 |
| logic 3 | 11 | 10 | — |
| permutations | 14 | 20 | 21 |

**CS: context sensitive rewriting.** Although other participants provide support for very sophisticated strategies, Maude is the only system among the participants supporting local strategies *à la* OBJ.

*Sieve of Eratosthenes.* The specification of the sieve of Erathostenes algorithm is used to compute the first 20, 100, and 1,000 prime numbers.

| | Maude32 | Maude64 |
|---|---|---|
| 20 | 2 | 2 |
| 100 | 152 | 125 |
| 1,000 | 165,039 | 135,639 |

## 5 The TIL chairmarks

In addition to the problems used in the previous competition (see Section 4), we included a few transformation problems from the TIL Chairmarks, by J. Cordy and E. Visser. Detailed information on the TIL Chairmark is available in the web site at http://www.program-transformation.org/Sts/TILChairmarks. As Cordy and Visser explain in this web page, "They are called *chairmarks* because they are too small to be called *benchmarks*". From all the tests proposed there, we chose six of them, trying to cover different kinds of problems. Examples illustrating some of the transformations proposed are included here, see http://www.program-transformation.org/Sts/TILChairmarks for examples and additional explanations on the rest, and also for the rest of the transformations proposed.

The problems chosen, with the numbers as in the TIL Chairmarks site, are:

**2.2 For to whiles:** This transformation restructures all for-loops in a TIL program to their while equivalents. Figure 4 shows an example of the application on this transformation to a TIL program.

```
for i := 1 to 9 do
   for j := 1 to 10 do
       write i * j;
   end
end
```

```
 var i;
i := 1;
while i != 9 + 1 do
    var j;
     var j;
    j := 1;
    while j != 10 + 1 do
        write i * j;
        j := j + 1;
    end
    i := i + 1;
end
```

**Fig. 4.** Program that outputs the first 10 multiples of numbers 1 through 9. The program in the right-hand side is the result of applying transformation 2.2 to the program on the left.

**2.4 Declarations to local:** Declaration are moved to its most local context.
**3.2 Common subexpression elimination:** Common subexpressions are recognized and factored out to new temporary variables.
**4.1 Redundant declarations:** Unused declarations are detected and removed.
**4.2 Statistics:** The number of statements of different kinds (declarations, assignments, ifs, whiles, fors, reads, and writes) in a program are counted.
**5.1 Interpretation:** TIL programs are executed by source transformation/rewriting.

Notice that, although clearly stated, the problems can be solved in different ways, and the outputs given in different forms. The outputs were not sys-

tematically checked. The outputs given and the program transformations proposed by the different systems are available at the competition's web site at http://www.lcc.uma.es/rewriting_competition. Given the interpreter provided as solution of the task 5.1, we can at least think of checking that both programs give the same result. But it was not done in this edition.

In ASF+SDF, implementing Tasks 2.2, 4.1 and 4.2 is straightforward. Task 2.4 requires a way of swapping statements; once it is clear how this should be done, the solution can be specified quite easily. The algorithmics needed to solve Task 3.2 are not trivial. Indeed, in the case of ASF+SDF most of the time was spent on implementing this 'chairmark'. Finally, the interpreter for Task 5.1 would take some time to implement without prior experience, but there exists an interpreter specified using ASF+SDF for a similar imperative language that can be used to understand the general idea behind such an interpreter.

For Maude the situation is very similar to the one for ASF+SDF. In this case, all the experience gathered along the years in giving semantics and defining execution environments for different languages is of great help.

Stratego appears to be a suitable language for the implementation of the TIL chairmarks. Simple transformations like 2.2 are defined with help of rewrite rules that are applied in a traversal strategy. This can be a general traversal strategy like topdown (2.2), or a custom traversal (for example 5.1). The separation of rules and strategies enables reuse. An example of reuse can be found in Task 4.2 where the *occurrences* strategy is used to collect statistic data. Sometimes the application of a rewrite rule depends on contextual information. Context information is handled with help of dynamic rules which are created during a traversal and can be scoped. Dynamic rules have been specifically designed for concisely handling problems as seen in the chairmarks, making Stratego highly effective at solving these problems. Dynamic rules are used in Tasks 2.4, 3.2, and 4.1 to implement lookup tables for variables and declarations.

The TIL chairmarks are typical applications for the Tom system. By using ANTLR it was straightforward to implement a parser. Then, given the produced AST, Tom appeared very appropriate to describe and implement the various transformations and optimizations: we have used the notion of rule (elementary strategy) to describe the transformations, and the user defined strategy language to describe how to apply the rules. E.g, Task 3.2 was solved using two strategies and a Java HashMap; Task 4.1 was solved, in less than 100 lines, using two strategies (one parameterized by a String) and a topdown Task 4.2 was solved, in around 70 lines, using a *count* strategy, integer counters and a topdown.

The TIL source transformation tasks are the kind of problems that TXL was designed for, and all of them are relatively straightforward for an experienced TXL programmer as self-contained TXL programs with no need for external tools or support routines. Task 4.1 is a single rewrite rule of 9 lines in TXL's vertical rule layout, using a scoped searching guard. Task 2.4 in TXL uses a sorting strategy in two parts, moving declarations to the first statement that uses them, and then moving them inside if it is a compound statement, using about 100 lines. Task 3.2 is a bit more challenging, using TXL rule parameters and scoped

application to find and replace subexpressions with a searching guard to insure non-interference, for a total of 80 lines. Task 4.2 exploits the TXL built-in type extract and count rules to solve the problem in one rule of 37 lines. Finally, the full TIL interpreter in TXL (Task 5.1) uses a pure rewriting interpretation with global terms to store the state, taking 286 lines.

Given the facilities provided by the different systems and the simplicity of most of the tasks, the tasks were solved in a short time, being most of the time spent in designing the solutions and debugging them.

## 6 Conclusions

As in the previous Rewrite Engines Competitions, we believe that both rewrite engines users and developers have benefited from this third edition of the competition. Although in edition we took a great step forward, by having five systems, focusing on program transformations without forgetting performance, and on automation, there is still a lot to be done towards having a real competition and really showing the potential of all the participating systems. In any case, our main goals were satisfied: we got to know each of the systems better, some of the strengths and weaknesses of the engines were shown, and we got more motivation to go on working on our respective systems.

And one wish for the competition: More automatization is required! For entering the programs, time capturing, results table generation, etc.

## Acknowledgements

We thank P. Ölveczky, as organizer of WRLA 2010, for inviting us to organize the competition, and to G. Roşu for getting the ball rolling in the 1st REC. And, of course, we have to thank all the people who have participated in the development of all the rewrite engines in the competition. F. Durán and M. Roldán have been supported by Research Projects TIN2008-03107 and P07-TIC-03184.

## References

1. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In F. Baader, ed., *Rewriting Techniques and Applications* , vol. 4533 of *Lecture Notes in Computer Science*, pp. 36–47. Springer, 2007.
2. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.
3. M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in Maude 2.4. In R. Treinen, ed., *Rewriting Techniques and Applications (RTA'09)*, vol. 5595 of *Lecture Notes in Computer Science*, pp. 380–390. Springer, 2009.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, vol. 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
6. J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.
7. J. R. Cordy, C. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
8. T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Agile parsing in TXL. *Automated Software Engineering*, 10(4):311–336, 2003.
9. G. Denker and C. Talcott, eds.. *6th Intl. Workshop on Rewriting Logic and its Applications*, vol. 176. Elsevier, 2007.
10. G. Denker, C. Talcott, G. Roşu, M. van den Brand, S. Eker, and T. F. Şerbănuţă. Rewriting logic systems. *Electronic Notes in Theoretical Computer Science*, 176(4):233–247, 2007.
11. F. Durán, M. Roldán, E. Balland, M. van den Brand, S. Eker, K. T. Kalleberg, L. C. L. Kats, P.-E. Moreau, R. Schevchenko, and E. Visser. The second rewrite engines competition. In G. Roşu, ed., *Procs. 7th Intl. Workshop on Rewriting Logic and its Applications (WRLA'08)*, vol. 238 of *Electronic Notes in Theoretical Computer Science*, pp. 281–291. Elsevier, 2008.
12. P. Klint, T. van der Storm, and J. Vinju. RASCAL: a domain specific language for source code analysis and manipulation. In *9th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, pp. 168–177, 2009.
13. G. Roşu, ed.. *Procs. 7th Intl. Workshop on Rewriting Logic and its Applications (WRLA'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.
14. R. Shevchenko and A. Doroshenko. A rewriting framework for rule-based programming dynamic applications. *Fundamenta Informaticae*, 72(1–3):95–108, 2006.
15. A. Sloane. Experiences with domain-specific language embedding in Scala. In J. Lawall and L. Reveillere, eds., *Procs. of the 2nd Intl. Workshop on Domain-Specific Program Development*, 2008.
16. M. Tomita. LR parsers for natural languages. In *Procs. of the 10th Intl. Conf. on Computational Linguistics and 22nd annual meeting of Assoc. for Computational Linguistics (ACL-22)*, pp. 354–357. Assoc. for Computational Linguistics, 1984.
17. M. van den Brand. Applications of the ASF+SDF meta-environment. In R. Lämmel, J. Saraiva, and J. Visser, eds., *Generative and Transformational Techniques in Software Engineering, Intl. Summer School (GTTSE'05)*, vol. 4143 of *Lecture Notes in Computer Science*, pp. 278–296. Springer, 2006.
18. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.
19. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.
20. M. G. J. van den Brand, A. van Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser & J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, ed., *Compiler Construction (CC'01)*, vol. 2027 of *Lecture Notes in Computer Science*, pp. 365–370. Springer, 2001.
21. A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.
22. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, ed., *Rewriting Techniques and Applications*, vol. 2051 of *Lecture Notes in Computer Science*, pp. 357–361. Springer, 2001.