# Environments for Term Rewriting Engines for Free!

Mark van den Brand[1,3], Pierre-Etienne Moreau[2], and Jurgen Vinju[1]

[1] CWI, Kruislaan 413,
NL-1098 SJ Amterdam, The Netherlands
{Mark.van.den.Brand,Jurgen.Vinju}@cwi.nl

[2] LORIA-INRIA, 615, rue du Jardin Botanique,
BP 101, 54602 Villers-lès-Nancy Cedex France
Pierre-Etienne.Moreau@loria.fr

[3] Vrije Universiteit, De Boelelaan 1081A,
NL-1081 HV Amsterdam, The Netherlands

**Abstract.** Term rewriting can only be applied if practical implementations of term rewriting engines exist. New rewriting engines are designed and implemented either to experiment with new (theoretical) results or to be able to tackle new application areas. In this paper we present the Meta-Environment: an environment for rapidly implementing the syntax and semantics of term rewriting based formalisms. We provide not only the basic building blocks, but complete interactive programming environments that only need to be instantiated by the details of a new formalism.

## 1 Introduction

Term rewriting can only be applied if practical implementations of term rewriting engines exist. New rewriting engines are designed and implemented either to experiment with new (theoretical) results or to be able to tackle new application areas, e.g., protocol verification, software renovation, etc. However, rewrite engines alone are not enough to implement real applications.

An analysis of existing applications of term rewriting, e.g. facilitated by formalisms like ASF+SDF [11], ELAN [3], MAUDE [9], RRL [14], STRATEGO [18], TXL [10], reveals the following four required aspects:

- a *formalism* that can be executed by a rewriting engine.
- *parsers* to implement the syntax of the formalism and the terms.
- a *rewriting engine* to implement the semantics of the formalism.
- a *programming environment* for supporting user-interaction.

A formalism introduces the syntactic notions that correspond to the operational semantics of the rewriting engine. This allows the user to write readable specifications. The parsers provide the connection from the formalism to the rewriting engine via abstract syntax trees. The programming environment can be either

a set of practical command line tools, an integrated system with a graphical user-interface, or some combination. It offers a user-interface tailored towards the formalism for interacting with the rewriting engine. For a detailed overview of rewriting-based systems we refer to [12].

Implementing the above four entities is usually a major research and software engineering effort, even if we target only small but meaningful examples. It is a long path from a description of a term rewriting engine, via language design for the corresponding formalism, to a usable programming environment.

In this paper we present the Meta-Environment: *An open architecture of tools, libraries, user-interfaces and code generators targeted to the design and implementation of term rewriting environments.*

We show that by using the Meta-Environment a mature programming environment for a new term rewriting formalism can be obtained in a few steps. Our approach is based on well-known software engineering concepts: standardization (of architecture and exchange format), software reuse (component based development), source code generation and parameterization.

### 1.1 Requirements

Real-world examples of term rewriting systems are to be found in many areas, including the following ([12]): rewriting workbenches, computer algebra, symbolic computation, functional programming, definition of programming languages, theorem proving, and generation, analysis, and transformation of programs.

These application areas are quite different, which explains the existence of several formalisms each tailored for a certain application domain. Each area influences the design and implementation of a term rewriting environment in several ways. We identify the following common requirements:

- *Openness.* Collaboration with unforseen components is often needed. It asks for an open architecture to facilitate communication between the environment, the rewriting engine, and foreign tools.
- *Readable syntax.* Syntax is an important design issue for term rewriting formalisms. Although conceptually syntax might be a minor detail, a formalism that has no practical and readable syntax is not usable.
- *Scalability.* Most real-world examples lead to big specifications or big terms. Scalability means that the implementation is capable of handling such problems using a moderate amount of resources.
- *Graphical User Interface.* A GUI with editors is needed. It automates as much of the browsing, editing and testing, of specifications as possible.

The above four issues offer no deep conceptual challenges, but still they stand for a considerable design and engineering effort. We offer immediately usable solutions concerning each of those issues in this paper. This paves the way for the application of new experiments concerning term rewriting that would otherwise have cost months to implement. In that sense, this paper contributes to the promotion and the development of rewriting techniques and their applications.

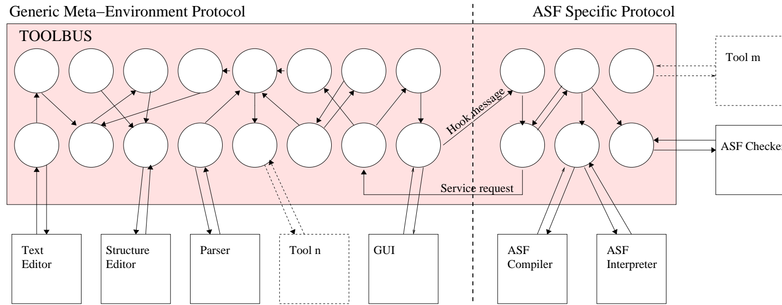


**Fig. 1.** A complete environment consisting of a generic part and an ASF specific part

## 2 Architecture for an open environment

In Section 3 we discuss the specific components of the Meta-Environment that can be used to implement a new term rewriting environment. An example environment is implemented in Section 4. Here we discuss the general architecture of the Meta-Environment.

The main issue is to separate computation from communication. This separation is achieved by means of a software coordination architecture and a generic data exchange format. An environment is obtained by plugging in the appropriate components into this architecture.

*ToolBus.* To prevent entangling of coordination with computation in components we introduce a software coordination architecture, the ToolBus [1]. It is a programmable software bus based on Process Algebra. Coordination is expressed by a formal description of the cooperation protocol between components while computation is expressed inside the components that may be written in any language. Figure 1 visualizes a ToolBus application (to be discussed below).

Separating computation from communication means that each of these components is made as *independent* as possible from the others. Each component provides a certain service to the other components via the software bus. They interact with each other using messages. The organization of this interaction is fully described using a script that corresponds to a collection of process algebra expressions.

ATerms. Coordination protocol and components have to share data. We use ATerms [5] for this purpose. These are normal prefix terms with optional annotations added to each node. The annotations are used to store tool-specific information such as text coordinates or proof obligations. All data that is communicated via the ToolBus is encoded as ATerms. ATerms are comparable to XML, both are generic data representations. Although there are tools for conversions between these formats, we prefer ATerms for efficiency reasons. They can be linearized using either a readable representation or a very dense binary encoding.

ATERMS can not only be used as a generic data exchange format but also to implement an efficient term data structure in rewriting engines. The ATERM library offers a complete programming interface to the term data structure. It is used to implement term rewriting interpreters or run-time support for compiled rewriting systems. The following three properties of the ATERM library are essential for term rewriting:

- Little memory usage per node.
- Maximal sub-term sharing.
- Automatic garbage collection.

Maximal sharing has proven to be a very good method for dealing with large amounts of terms during term rewriting [4, 18]. It implies that term equality reduces to *pointer equality*. Automatic garbage collection is a very practical feature that significantly reduces the effort of designing a new rewriting engine or compiler.

*Meta-Environment Protocol.* The ToolBus and ATERMS are more widely applicable than just for term rewriting environments. To instantiate this generic architecture, the Meta-Environment ToolBus scripts implement a coordination protocol between its components. Together with the tools, libraries and program generators this protocol implements the basic functionality of an interactive environment.

The Meta-Environment protocol makes no assumptions about the rewriting engine and its coordination with other tools. In order to make a complete term rewriting environment we must complement the generic protocol with specific coordination for every new term rewriting formalism.

For example, the architecture of the ASF+SDF Meta-Environment is shown in Figure 1. The ToolBus executes the generic Meta-Environment protocol, depicted by the circles in the left-hand side of the picture. It communicates with external tools, depicted by squares. The right-hand side of the picture shows a specific extension of the Meta-Environment protocol, in this example it is designed for the ASF+SDF rewriting engines. It can be replaced by another protocol in order to construct an environment for a different rewriting formalism.

*Hooks.* The messages that can be received by the generic part are known in advance, simply because this part of the system is fixed. The reverse is not true, the generic part can make no assumptions about the other part of the system.

We identify messages that are sent from the generic part of the Meta-Environment to the rewriting formalism part as so-called *hooks*. Each instance of an environment should *at least* implement a receiver for each of these hooks. Table 1 shows the basic Meta-Environment hooks. The first four hooks instantiate parameters of the GUI and the editors. The last four hooks are events that need to be handled in a manner that is specific for the rewriting formalisms.

| Hook | Description |
| --- | --- |
| `environment-name(Name)` | The main GUI window will display this name |
| `extensions(Sig, Sem, Term)` | Declares the extensions of different file types |
| `stdlib-path(Path)` | Sets the path to a standard library |
| `semantics-top-sort(Sort)` | Declares the top non-terminal of a specification |
| `rewrite(Sig, Sem, Term)` | Rewrite a term using a specification |
| `pre-parser-generation(Sig)` | Manipulate the syntax before parser generation |
| `rename-semantics(Sig,Binds,Sem)` | Implement module parameterization |
| `pre-rewrite(Sig,Spec)` | Actions to do before rewriting |

**Table 1.** The Meta-Environment hooks: the hooks that parameterize the GUI (top half), and events concerning the syntax and semantics of a term rewriting formalism (bottom half).

## 3 Reusable components

In this section we present reusable components to implement each aspect of the design of a term rewriting environment. The components are either tools, libraries or code generators. In Section 4 we explain how to use these components to create a programming environment using an example term rewriting formalism.

### 3.1 Generalized Parsing for a readable formalism

We offer generic and reusable parsing technology. An implementation of parsing usually consists of a syntax definition formalism, a parser generator, and run-time support for parsing. Additionally, automated parse tree construction and abstract syntax tree construction are offered. Table 2 shows a list of components related to parsing.

*Syntax.* SDF is a declarative syntax definition formalism used to define modular context-free grammars. Both lexical syntax and context-free syntax can be expressed in a uniform manner. Among other disambiguation constructs, notions for defining associativity and relative priority of operators are present.

Furthermore, SDF offers a simple but effective parameterization mechanism. A module may be parameterized by formal parameters attached to the module name. Using the import mechanism of SDF this parameter can be bound to an actual non-terminal.

Programs that deal with syntax definitions can use the SDF library. It provides a complete high-level programming interface for dealing with syntax definitions.

*Concrete syntax.* Recall that a syntax definition can serve as a many-sorted signature for a term rewriting system. The grammar productions in the definition are the operators of the signature and the non-terminals are the sorts.

| Tool | Type | | Description |
|---|---|---|---|
| pgen | SDF | → Table | Generates a parse table from a syntax definition |
| sglr | Table×Str | → AsFix | parses an input string and yields a derivation |
| implode | AsFix | → ATerm | Maps a parse tree to an abstract term |
| posinfo | AsFix | → AsFix | Adds line and column annotations |
| unparse | AsFix | → Str | Yields the string that is derived by a parse tree |

**Table 2.** A list of the most frequently used components for SDF and ASFIX

The number of non-terminals used in a grammar production is the arity of an operator.

Concrete syntax for any term rewriting formalism can be obtained by simply expressing both the fixed syntax of the formalism and the user defined syntax of the terms in SDF. A parameterized SDF module is used to describe the fixed syntax. This module can be imported for every sort in the user-defined syntax. An example is given in Section 4.

*SGLR.* To implement the SDF formalism, we use scannerless generalized LR parsing [7]. The result is a simple parsing architecture, but capable of handling any modular context-free grammar.

ASFIX. SGLR produces parse trees represented as ATERMS. This specific class of ATERMS is called ASFIX. Every ASFIX parse tree explains exactly, for each character of the input, which SDF productions were applied to obtain a derivation. A library is offered to be able to create components that deal with ASFIX.

### 3.2 Establishing the connection between parsing and rewriting

The SDF library and the ASFIX library can be used to implement the connection between the parser and a rewriting engine. Furthermore, we can also automatically generate new libraries specifically tailored towards the rewriting formalism that we want to implement [13].

We use an SDF definition of the new formalism to generate `C` or `Java` libraries that hide the actual ATERM representation of a parse tree of a specification behind a typed interface. The generated interfaces offer: reading in parse trees, constructors, getters and setters for each operator of the new formalism. Apart from saving a lot of time, using these code generators has two major advantages:

- The term rewriter can be developed at a higher level of abstraction.
- Programming errors are prevented by the strictness of the generated types.

### 3.3 Graphical User Interface

*MetaStudio.* The Meta-Environment contains a user-interface written in Java (Figure 2). It can be used to browse modules. Every module has a number of actions that can be activated using the mouse. The actions are sent to the ToolBus. MetaStudio has parameters to configure the name of the environment, the typical file extensions, etc.

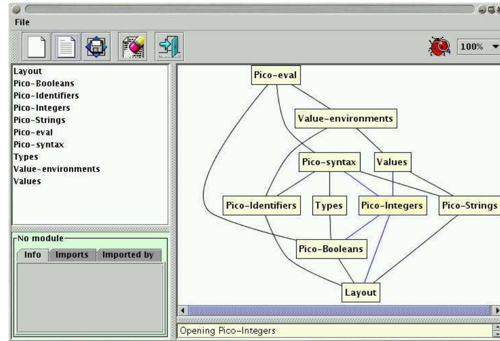


**Fig. 2.** GUI of the Meta-Environment displaying an import relation.

*Editors.* Editing of specifications and terms is done via `XEmacs`[1]. To implement structure-editing capabilities, `XEmacs` communicates with another component that holds a tree representation of the edited text.

*Utilities.* Among other utilities, we offer file I/O and in-memory storage that aid in the implementation of an interactive environment.

## 4 A new environment in a few steps

In this section we show the steps involved in designing a new environment. We take a small imaginary formalism called "RHO" as a running example. It is a subset of the $\rho$-calculus [8], having first-class rewrite rules and an explicit application operator. The recipe to create a RHO environment is:

1. Instantiate the parameters of the GUI.
2. Define the syntax of RHO.
3. Write some small RHO specifications.
4. Implement and connect a RHO interpreter.
5. Connect other components.

*1. Instantiate the parameters of the GUI:* We start from a standard ToolBus script that implements default behavior for all the hooks of Table 1. We can immediately bind some of the configuration parameters of the GUI. In the case of RHO, we can instantiate two hooks: `environment-name("The RHO Environment")` and `extensions(".sdf",".rho",".trm")`.

Using the RHO Meta-Environment is immediately possible. It offers the user three kinds of syntax-directed editors that can be used to complete the rest of the recipe: SDF editors, editors for the (yet unspecified) RHO formalism, and term editors.

[1] `http://www.xemacs.org`

```
module Rho[Term]
imports Rho-Lexical
exports
sorts Rho Decl Decls
context-free syntax
"rho" Decl*      -> Decls
Id ":=" Rho      -> Decl
Term | Id        -> Rho
Rho "->" Rho     -> Rho {right}
Rho "." Rho      -> Rho {left}
```

```
module Rho-Lexical
exports
lexical syntax
[A-Za-z][a-zA-Z0-9]*  -> Id
[\ \t\n]              -> LAYOUT
"%%" ~[\n]* "\n"      -> LAYOUT
```

**Fig. 3.** A parameterized syntax definition of the formalism RHO.

*2. Define the syntax of* RHO*:* Figure 3 shows how the SDF editors can be used to define the syntax of RHO[2]. It has some predefined operators like assignment (`":="`), abstraction (`"->"`) and application (`"."`), but also concrete syntax for basic terms. So, a part of the syntax of a RHO term is user-defined. The parameterization mechanism of SDF is used to leave a placeholder (`Term`) at the location where user-defined terms are expected[3]. The `Term` parameter will later be instantiated when writing RHO specifications.

To make the syntax-directed editors for RHO files work properly we now have to instantiate the following hook: `semantic-top-sort("Decls")`. The parameter `"Decls"` refers to the top sort of the definition in Figure 3.

*3. Write some small* RHO *specifications:* We want to test the syntax of the new formalism. Figure 4 shows how two editors are used to specify the signature and some rules for the Boolean conjunction. Notice that the `Rho` module is imported explicitly by the `Booleans` module, here we instantiate the `Term` placeholder for the user-defined syntax. In Section 4 we explain how to add the imports implicitly.

We can now experiment with the syntax of RHO, define some more operators, basic data-types or start a standard library of RHO specifications. For the GUI, the location of the library should be instantiated using the `stdlib-path` hook.

*4. Implement and connect a* RHO *interpreter:* As mentioned in Section 2, the ATERM library is an efficient choice for a term implementation. Apart from that we present the details of the connection between a parsed specification and an implementation of the operational semantics of RHO. The algorithmic details of evaluating RHO are left to the reader, because that changes with each instance of a new formalism.

The `rewrite` hook connects a rewriting engine to the RHO environment: `rewrite(Syntax,Semantics,Term)` From this message we receive the information that is to be used by the rewriting engine. Note that this does not prohibit to request any other information from other components using extra messages. The input data that is received can be characterized as follows: `Syntax` is a list of

[2] For the sake of brevity, Figure 3 does not show any priorities between operators.
[3] Having concrete syntax of terms is not obligatory.

```
emacs: Booleans.sdf
File Edit Mule Apps Options Buffers Tools Meta-Environment
module Booleans
imports Rho[Bool]
exports sorts Bool
context-free syntax
  "true" | "false" -> Bool
  Bool "&" Bool    -> Bool {left}
variables "B"      -> Bool
ISO8--**-XEmacs: Booleans.sdf       (Fundament
Focus symbol: ModuleName
```

```
emacs: Booleans.rho
File Edit Mule Apps Options Buffers Tools Meta-Environment
rho

conj1 := true & B  -> B
conj2 := false & B -> false

test  := conj1 . true & false
ISO8-----XEmacs: Booleans.rho       (Fundament
Focus symbol: Rho
```

**Fig. 4.** A definition of the Boolean conjunction in SDF+RHO.

all SDF modules (the parse-trees of `Rho.sdf` and `Booleans.sdf`). `Semantics` is a list of all RHO modules (the parse-tree of `Booleans.rho`). `Term` is the expression that is to be normalized (for example a parse-tree of a file called `test.trm`).

Two scenarios are to be considered: either a RHO engine already exists, or a new engine has to be designed from scratch. In the first case, the data-types of the Meta-Environment will be converted to the internal representation of the existing engine. In the second case, we can implement a new engine based on the data-types of the Meta-Environment directly. In both cases the three data-types of the Meta-Environment are important: SDF, ASFIX and ATERMS. The libraries and generators ensure that these cases can be specified on a high level of abstraction. We split the work into the signature and semantics parts of RHO.

*Signature.* To extract the needed information from the user-defined signature the SDF modules should be analyzed. The SDF library is the appropriate mechanism to inspect them in a straightforward manner.

*Semantics.* Due to having concrete syntax, the list of parse trees that represent RHO modules is not defined by a fixed signature. We can divide the set of operators in two categories:

- *A fixed* set of operators that correspond to the basic operators of the formalism. Each fixed operator represents a syntactical notion that should be given a meaning by the operational semantics. For RHO, assignment, abstraction, and application are examples of fixed operators.
- *Free* terms occur at the location where the syntax is user-defined. In RHO this is either as the right-hand side of an assignment or as a child of the abstraction or application operators.

There is a practical solution for dealing with each of these two classes of operators. Firstly, from an SDF definition for RHO we generate a library specifically tailored for RHO. This library is used to recognize the operators of RHO and extract information via an abstract typed interface. For example, one of the C function headers in this generated library is: `Rho getRuleLhs(Rho rule)`. A RHO interpreter can use it to retrieve the left-hand side of a rule.

Secondly, the free terms can be mapped to simple prefix ATERMS using the component `implode`, or they can be analyzed directly using the ASFIX library. The choice depends on the application area. E.g., for source code renovation

details such as white space and source code comments are important, but for symbolic computation this information might as well be thrown away in favor of efficiency.

In the case of an existing engine, the above interfaces are used to extract information before providing it to the engine. In the case of a new engine, the interfaces are used to directly specify the operational semantics of RHO.

*5. Connect other components:* There are some more hooks that can be instantiated in order to influence the behavior of the Meta-Environment. Also, the RHO part of the newly created environment might introduce other components besides the rewriting engine.

We give two examples here. The `pre-parser-generation` hook can be used to extend the user-defined syntax with imports of the RHO syntax automatically for each non-terminal. Secondly, the `pre-rewrite` hook hook can be used to connect an automatic verifier or prover like a Knuth-Bendix completion procedure.

Adding unanticipated tools is facilitated at three levels by the Meta-Environment. Firstly, an SDF production can have any attribute to make it possible to express special properties of operators for the benefit of new tools. An example: `B "&" B -> B { left, lpo-precedence(42) }`. Secondly, any ATERM can be annotated with extra information without affecting the other components. For example: `and(true,false){not-reduced}`. Finally, all existing services of the Meta-Environment are available to the new tool. It can for example open a new editor to show its results using this message: `new-editor(Contents)`

## 5 Instantiations of the Meta-Environment

We now introduce the four formalisms we have implemented so far using the above recipe. We focus on the discriminating aspects of each language.

ASF [11] is a term rewriting formalism based on leftmost-innermost normalization. The rules are called equations and are written in concrete syntax. Equations can have a list of conditions which must all evaluate to true before a reduction succeeds. The operational semantics of ASF also introduces *rewriting with layout* and *traversal functions* [6], operators that traverse the subterm they are applied to.

The above features correspond to the application areas of ASF. It is mainly used for design of the syntax and semantics of domain specific languages and analysis and transformation of programs in existing programming languages. From the application perspective ASF is an expressive form of first-order functional programming. The Meta-Environment serves as a programming environment for ASF.

ELAN [3] is based on rewrite rules too. It provides a strategy language, allowing to control the application of rules instead of leaving this to a fixed normalization strategy. Primitive strategies are labelled rewrite rules, which can be combined

using strategy basic operators. New strategy operators can be expressed by defining them in terms of less complex ones. ELAN supports the design of theorem provers, logic programming languages, constraint solvers and decision procedures and offers a modular framework for studying their combination.

In order to improve the architecture, and to make the ELAN system more interactive, it was decided to redesign the ELAN system based on the Meta-Environment. The instantiation of the ELAN environment involved the implementation of several new components, among others an interpreter. Constructing the ELAN environment was a matter of a few months.

*The $\rho$-calculus* [8] integrates in a uniform and simple setting first-order rewriting, lambda-calculus and non-deterministic computations. Its abstraction mechanism is based on the rewrite rule formation. The application operator is explicit, allowing to handle sets of results explicitly.

The $\rho$-calculus is typically a new rewriting formalism which can benefit from the the Meta-Environment. We have prototyped a workbench for the complete $\rho$-calculus. After that, we connected an existing $\rho$-calculus interpreter. This experiment was realized in one day.

*The JITty interpreter* [17] is a part of the $\mu$CRL [2] toolset. In this toolset it is used as an execution mechanism for rewrite rules. JITty is not supported by its own formalism or a specialized environment. However, the ideas of the JITty interpreter are more generally applicable. It implements an interesting normalization strategy, the so-called `just-in-time` strategy. A workbench for the JITty interpreter was developed in a few hours that allowed to perform experiments with the JITty interpreter.

## 6 Conclusions

Experiments with and applications of term rewriting engines are within much closer reach using the Meta-Environment, as compared to designing and engineering a new formalism from scratch.

We have presented a generic approach for rapidly developing the three major ingredients of a term rewriting based formalism: syntax, rewriting, and an environment. Using the scalable technology of the Meta-Environment significantly reduces the effort to develop them. We used our approach to build four environments. Two of them are actively used by their respective communities. The others serve as workbenches for new developments in term rewriting.

The Meta-Environment and its components can now support several term rewriting formalisms. A future step is to build environments for languages like Action Semantics [15] and TOM [16]. Apart from more environments, other future work consists of even further parameterization and modularization of the Meta-Environment. Making the Meta-Environment open to different syntax definition formalisms is an example. The Meta-Environment can be downloaded via: `http://www.cwi.nl/projects/MetaEnv`

## References


1. J. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.
2. S. Blom, W. Fokkink, J. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. $\mu$CRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer-Verlag, 2001.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *WRLA*, volume 15 of *ENTCS*. Elsevier Sciences, 1998.
4. M. v. d. Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.
5. M. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.
6. M. v. d. Brand, P. Klint, and J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, Centrum voor Wiskunde en Informatica, 2001.
7. M. v. d. Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In R. N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.
8. H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.
9. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *First international workshop on rewriting logic*, volume 4, Asilomar (California), 1996. ENTCS.
10. J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
11. A. v. Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.
12. J. Heering and P. Klint. *Term Rewriting Systems*, chapter 15, pages 787–808. Cambridge University Press, 2003.
13. H. d. Jong and P. Olivier. Generation of abstract programming interfaces from syntax definitions. Technical Report SEN-R0212, Centrum voor Wiskunde en Informatica (CWI), Aug. 2002.
14. D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.
15. S. Lassen, P. Mosses, and D. Watt. An introduction to AN-2, the proposed new version of Action Notation. In *Proc. 3rd International Workshop on Action Semantics*, volume NS-00-6 of *Notes Series*, pages 19–36. BRICS, 2000.
16. P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
17. J. v. d. Pol. JITty: a Rewriter with Strategy Annotations. In S. Tison, editor, *Rewriting Techniques and Applications*, volume 2378 of *LNCS*, pages 367–370. Springer-Verlag, 2002.
18. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.