Electronic Notes in Theoretical Computer Science 15 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume15.html 23 pages

Term Rewriting for Sale

M.G.J. van den Brand

Department of Software Engineering
Centrum voor Wiskunde en Informatica (CWI)
and
Programming Research Group
University of Amsterdam
Amsterdam, The Netherlands

P. Klint

Department of Software Engineering
Centrum voor Wiskunde en Informatica (CWI)
and
Programming Research Group
University of Amsterdam
Amsterdam, The Netherlands

C. Verhoef

Programming Research Group
University of Amsterdam
Amsterdam, The Netherlands

Abstract

Term rewriting has a large potential for industrial applications, but these applications are always larger than one could ever dream of: huge sets of rewrite rules and gigantic terms to rewrite pose interesting challenges for implementors and theoreticians alike. We give a brief overview of the generation of term-rewriting-based tools as done in the ASF+SDF Meta-Environment and then we sketch two major applications of term rewriting: transformation of legacy COBOL systems and compilation of ASF+SDF to C. Based on these experiences we suggest the study of topics that could further advance the use of term rewriting in industrial applications: persistent term databases, generalized LR parsing *versus* parallel term rewriting, and coordination languages *versus* strategy languages. It will turn out that we have an "alien" view on research in term rewriting: properties like confluence and termination are of very limited use when selling term rewriting to industry.

1 Background

Term rewriting is omnipresent in computer science. It has been used to formalize the notion of computability, to study computational procedures, to implement functional languages, to analyze and implement abstract data types, to decide word problems, to proof theorems, and so on [40]. In the early eighties an ESPRIT project was started to support the generation of interactive programming environments from language specifications using term rewriting as one of the basic implementation paradigms [29]. This research has resulted in the design of the Asf+Sdf formalism and its support environment, the Asf+Sdf Meta-Environment. ASF stands for Algebraic Specification Formalism [6]. It is an implementation of conditional term rewriting [36,8] possibly containing negative premises [37,45,46,26]. It also uses a form of prioritized rewriting [3,44]: there are default rewrite rules that apply if other rewrite rules do not match. SDF stands for Syntax Definition Formalism [28]. In fact, SDF is the part to conveniently define the signature Σ of a term rewriting system and in ASF this signature is used to specify the rewrite rules R. Terms are many-sorted and varyadic constructor functions are available for defining lists.

The formalism Asf+Sdf and the Asf+Sdf Meta-Environment have been described in a variety of papers and books [5,38,21]. In the context of this paper, the most significant properties of the Asf+Sdf technology are:

- Asf+Sdf is based on algebraic specifications and term rewriting is used for its implementation. Specifications consist of modules with conditional equations.
- Concrete syntax and abstract syntax are closely integrated: there is a standard mapping between the textual (concrete) representation of terms and their (abstract) representation as trees.
- The definition of concrete syntax is based on unrestricted context-free grammars, thus permitting modular composition of grammars. This is possible because of the parsing technology we are using, namely Generalized LR parsing [42,55,49,56].
- The Asf+Sdf Meta-Environment is highly incremental and supports the rapid development and prototyping of specifications.
- The data exchange format between components is based on terms. The format is called ATF (Annotated Term Format) [12]. Its implementation is called ATerms [11]. The data exchange between components of the Asf+Sdf Meta-Environment is called AsFix.

The Asf+Sdf view on the world is *grammar-centric* and can be characterized by the slogan

Every expression has a grammar, so it can be represented as a term.

All languages are treated equal, be it the language of truth values or the programming language COBOL. Simple Boolean expressions and COBOL programs are all treated as terms. The major surprise is that this paradigm scales up so well to large industrial applications, like:

- analysis and transformation tools for the renovation of COBOL programs [16,17,15,51,54,22,23];
- tools for the migration between COBOL dialects [15];
- reengineering tools for a proprietary (telecommunications) language [52];
- design of and tool support for a new, domain-specific language, called Risla, for financial applications [2,20];
- a code generator for the UML design notation;
- tool support for a specification language for railway safety [27,25].

An overview of industrial applications of the Asf+Sdf Meta-Environment can be found in [9]. Other large, non-commercial, applications of the Asf+Sdf technology are:

- specification, prototyping and final implementation of various components of the Asf+Sdf Meta-Environment itself, such as, a compiler [14], parser generator [56], and pretty printer generator [19];
- specification and prototyping of the ToolBus coordination architecture [7];
- tool support for action semantics [24];
- tool support for the CASL formalism [10].

In Section 2 we briefly discuss issues that have to be taken into account when cooperating with IT industry. Next, we sketch the global approach we use for the automatic generation of tools and applications (Section 3). Then, we highlight two major applications: the construction of analysis and transformation tools for COBOL (Section 4) and the Asf+Sdf compiler, which is written in Asf+Sdf (Section 5). Based on these experiences we suggest in Section 6 the study of topics that could further advance the use of term rewriting in industrial applications: persistent term databases, generalized LR parsing *versus* parallel term rewriting, and coordination languages *versus* strategy languages. It will turn out that we have an "alien" view on research in term rewriting: properties like confluence and termination are of very limited use when selling term rewriting to industry.

2 Dealing with IT Industry

We advocate the use of formal techniques in a limited application area: the construction of grammar-centric tools based on term rewriting. This is a modest goal compared to solving software engineering problems *in general* by means of formal techniques. The latter requires major paradigm shifts and re-education of the users of these formal techniques. The former limits the use of formal techniques to the tools that are being developed and these tools can be fitted in traditional software engineering frameworks. Even with this limited ambition, we have to face serious technology transfer problems.

Capers Jones states that the main cause of slow technology transfer appears to be based in the normal human psychology: the human mind establishes a set of paradigms about methods, tools, and technologies. Once such a paradigm is established, all evidence to the contrary tends to be discarded. As a result, the time from first creation of an idea to widespread deployment is usually no less than 14 years [34]. Perhaps it is human to fail to understand concepts when they are approached from a different viewpoint, in a different language. This phenomenon applies to using term rewriting in the IT industry as well. It is, however, interesting that the speed of adoption for new products seems to be increasing [47].

We give an indication of the time we measured from first contacts between CWI/University of Amsterdam and IT industry until the moment that a solid relation with the begin of technology transfer was established.

- Work on the domain-specific language Risla has lead to solid cooperation and use in IT industry after five years.
- Work on software renovation factories for COBOL has lead to intensive contacts between IT industry and CWI/University of Amsterdam after three years of research.
- Work on a code generator for UML took over four years before an amalgamation between CWI/University of Amsterdam and the involved industrial party was ensued.
- Work on railway safety regulations took also over three years before solid relations within the industry were established.
- Cooperation between companies whose core business is reengineering does take less time before serious relations are established. The time between first contact and formal cooperation varies from a few months to a year. We assume that the Year 2000 problem is a catalyst.

Adequate functionality, robustness and efficiency are, of course, important for industrializing new technologies. In our experience, however, the key inhibitor is the long time needed for technology transfer, industrialization and commercialization. The danger being that either researchers or industry (or both!) loose their interest in the transfer. This topic is extensively discussed in the excellent textbook [33].

3 The ASF+SDF Approach to Tool Generation

In a first approximation, the ASF+SDF approach is based on the generation of tools as shown in Figure 1. Starting points are a *context-free grammar* defining the lexical, concrete, abstract and unparsing (prettyprinting) syntax of a language, *analysis and transformation rules* defining what information is to be extracted from programs and what changes are to be performed on them, and *coordination rules* defining how the analysis and transformation rules have to be glued together in the tool that is to be generated. These definitions are fed into a *tool generator* that produces the desired tool.

The *generated tool* takes source text in the defined language as input, applies the analysis and transformation rules to it as prescribed by the coordina-

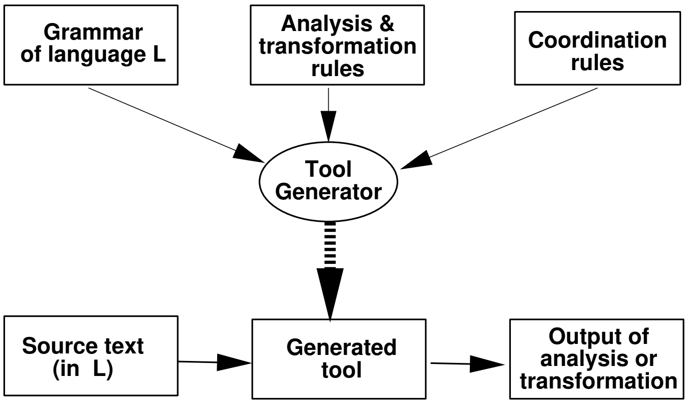

Fig. 1. Generating tools from language definitions.

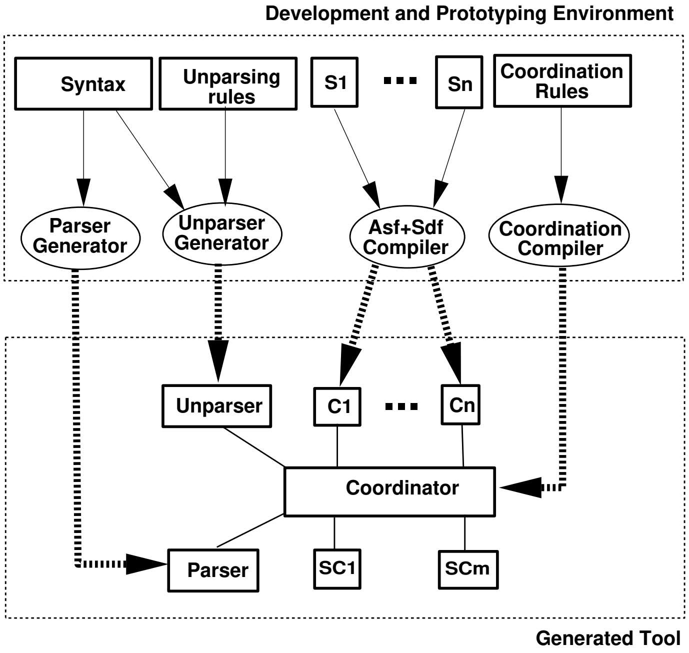

Fig. 2. The Asf+Sdf Meta-Environment.

tion rules and produces analysis results or complete (transformed) programs.

In a second approximation, the structure of the Asf+Sdf Meta-Environment is as shown in Figure 2. At the bottom we see, once more, a generated tool but now its internal structure is shown. At the top, we see an interactive development and prototyping environment in which specifications can be edited and executed. In older versions of the Asf+Sdf Meta-Environment the development and prototyping environment and the generated tools were closely intertwined. In newer versions, they are completely separated so that stand-alone tools can be generated.

Syntax, unparsing rules, and the specification modules $S_1, \ldots, S_n$ are all written in Asf+Sdf. For the coordination rules various formalism are in use (SEAL [41], ToolBus scripts [7]). The modules S_i are either created manually or they may be the result of a generation process (this is not further detailed in the figure).

The generation process consists of the following steps. The context-free grammar (possibly extended with unparsing rules) is transformed into a parser and unparser by, respectively, a parser generator and an unparser generator. The analysis and transformation rules are captured by specification modules $S_1, \ldots, S_n$. They are compiled to executable components (rewriting systems) $C_1, \ldots, C_n$ by the Asf+Sdf compiler. The coordination rules are compiled to an executable coordinator. The picture is completed by a number of standard components $SC_1, \ldots, SC_m$ that provide common services like persistent storage and syntax-directed editing. For instance, for the user-interface of the new Asf+Sdf Meta-Environment we use `wish` (a user-interface builder) in combination with `dot` (a graph drawing tool) as basic, off-the-shelf, components that can be effortlessly integrated in the open ToolBus architecture using the component-based software engineering philosophy described in [39].

4 Transforming COBOL Systems

Our first example of generated tools are renovation factories as needed for the analysis and remediation of legacy systems. Such factories are generated tools with the architecture as already shown in Figure 2. All the ingredients of such factories are generated from language specifications. Although our approach is completely generic, we will illustrate it here by way of examples for the COBOL language. We give now first a quick overview of the main ingredients of software renovation factories, then we discuss applications, and finally we give some measurements.

Parsing

Although the standardization of programming languages has a long history, it turns out that every new customer of a renovation factory brings its own language dialect, even for mainstream languages like COBOL. This is caused by the use of non-standard, compiler-specific, language features or even by the use of home-grown dialects. Mainstream (LR) parser generator technology breaks down in such cases since the continuous stream of grammar modifications needed to support all these dialects leads to unsurmountable maintenance problems.

These problems do not occur when Generalized LR parser technology is used. The key property of GLR parsers is that they can parse the complete class of context-free languages. As a result, the composition of two grammars (an essential operation for modular structuring) can again be parsed by a GLR parser. In other words, GLR parsing is essential for parsing modular grammars.

Modular grammars make it possible to place dialect specific parts of a language in separate modules. When we need them we can use them at will, without creating a maintenance problem. We can also easily add new grammar

```
Procedure-name-1.
  IF Condition-1
     Statement-1+
     GO TO Procedure-name-1.
```

Fig. 3. A native COBOL pattern.

rules to existing ones without having to solve conflicts: they are solved by the GLR algorithm.

In Section 6.2 we give a more detailed description of GLR parsing. Parsing technology in reengineering and its problems are further discussed in [18].

Unparsing

We also generate formatters from grammar descriptions. Given the grammar of a language (in SDF) an Asf+Sdf specification is generated that can be used as a default formatter for that language [19]. We adapt the generated Asf+Sdf specification to the demands of the company in question and we use the Asf+Sdf compiler to generate an efficient stand-alone unparser.

Patterns

We need a way to describe patterns for the recognition and transformation of code in a software renovation factory. The usual approach is to use *open terms* for this purpose: terms that represent a certain fixed pattern and contain variables at those positions where that pattern may vary.

A lesson that we learned from IT industry is that these patterns should be designed very carefully. The ideal situation is that the patterns for recognizing code resemble the code itself as much as possible.

Given the strong correspondence between concrete and abstract syntax in Asf+Sdf (and the use of GLR parsing, see below), we can implement this requirement quite effectively. Given a grammar (in SDF) for some language we generate a so-called *native pattern language* [54] for it. This contains, amongst others, appropriately named variable declarations for each language construct. An example of a native COBOL pattern is shown in Figure 3. As can be seen, the COBOL pattern uses all the keywords that are already available in the COBOL standard [1]. Variables like `procedure-name-1` and `Statement-1+` act as placeholders in the pattern. For more details we refer to [54].

The generation of these patterns is facilitated by the use of GLR parsing. In [50] it is observed that grammars are often structured to permit ease of parsing and not ease of conceptual understanding, i.e., they are parser-oriented. In many cases, the grammar has to be expressed in an unnatural form to satisfy the restrictions that the parser generator imposes, such as one token look-ahead, and certain conflicts during parsing (see Section 6.2 or [43]). This same observation was made in the design of SDF [28].

```
foo_Paragraph(
Procedure-name-1.
  IF Condition-1
     Statement-1+
     GO TO Procedure-name-1.)
=
Procedure-name-1.
  PERFORM UNTIL Condition-1
    Statement-1+
  END-PERFORM.
```

Fig. 4. Example transformation that eliminates a `GO TO`.

The conclusion that one can draw from these observations is that it is a bad idea to use such a parser-oriented grammar for constructing other functionality like, e.g., the generation of a native pattern language. Since we use GLR parsing technology, there are no parser-related restrictions on the form of the grammar rules, we can write “natural” grammar rules, and therefore, the underlying structure of the native patterns is natural as well.

Analysis and Transformation

Once we have patterns, we want to use them for the purpose of analysis or transformation of programs. The usual approach in term rewriting is to define analysis or transformation functions by means of rewrite rules. However, the number of rules required is directly related to the size of the signature of the language involved. In the case of real languages, e.g., COBOL, this becomes prohibitive.

To overcome this problem, we *generate* for a given grammar generic analysis and transformation functions [15] that can be specialized for different applications. For each sort in the grammar we generate a separate analysis and traversal function. The names of these functions are derived from the sort name used in the grammar. Here, again, we generate functionality that is as natural as possible for the people who need to work with these generated analysis and transformation functions in a renovation factory. As default behaviour, the generic analysis function performs a generic calculation for each subtree encountered, whereas the generic transformation function performs the identity transformation. This default behaviour can be easily redefined for each sort of subtree. In this way, it is only necessary to define *what* should happen for certain subtrees independently of *where* they occur.

An example may clarify the issues involved. Suppose we want to change patterns of the form shown in Figure 3 into more structured COBOL fragments not containing `GO TO` statements. For this purpose we want to define a traversal function `foo`. Our generator for analysis and transformation functions will produce for each sort `S` in the COBOL grammar the syntax and (transformation) semantics for the function `foo_S`. Its default behaviour is the identity transformation. In our example we only need to specify the be-

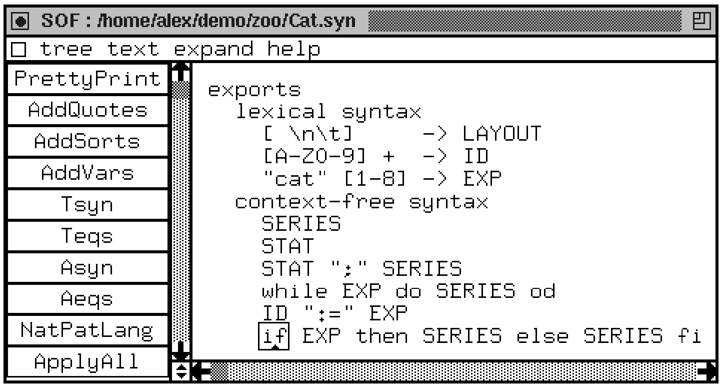

Fig. 5. Development environment for renovation factories.

haviour on COBOL paragraphs (e.g., `foo_Paragraph`) as is done in Figure 4. This simple transformation will transform all `GO TO` statements that are in fact while loops. The idea is that only for the sort `Paragraph` there is non-default behaviour: a change should only be made when this specific `GO TO` pattern is matched. While loops are represented in COBOL by way of the `PERFORM UNTIL` statement. Elimination of `GO TO` statements in COBOL is further described in [17].

Computer-aided Language Engineering

Since context-free grammars form the starting point of our approach, we heavily depend on the availability of grammars of very high quality. It is not unusual in industrial settings that grammars are completely absent, that they are incomplete or that they are of a poor quality. Not surprisingly, we have therefore also created an entire factory dedicated to the recovery, analysis and improvement of grammars themselves. Recall our slogan that "every expression has a grammar, so it can be represented as term", and observe that a grammar has also a grammar, so a grammar can be represented as a term as well. Such terms can be transformed and analyzed. We call this reengineering approach to grammars *computer aided language engineering* (CALE). It includes development tools for grammars, tools to assess the quality of existing grammars (completeness, metrics), and tools for generating grammars from other sources, such as language reference manuals or the source code of a compiler [52,53].

The productivity gains using these tools can be considerable. On the one hand, we measured a productivity of 300 production rules per month for writing grammars by hand [16]: this involved the manual extraction of a COBOL grammar from language reference manuals. On the other hand, we measured a productivity of 3000 (generated) production rules using a CALE factory in one afternoon [53]: this involved the automatic extraction of the grammar for a language used by a telecommunications company from the source code of a compiler for that language. Obviously, it is no longer possible to process such large grammars without appropriate tool support.

Creating renovation factories

Once we have a good grammar and have generated all the generic support we discussed, we start constructing transformations and analysis functions to set up assembly lines that solve reengineering problems. The important issue here is that we design small but effective components that can be easily reused. Currently, we test assembly lines interactively using a coordination architecture called SEAL [41] as can be seen in Figure 5. The code shown is a small example grammar. The buttons are dealing with the generation of the native pattern language (`NatPatLang`) and the generation of syntax and semantics of analysis (`Asyn`, `Aeqs`) and transformation support (`Tsyn`, `Teqs`). Figure 5 is also an example of how assembly lines look like during construction.

Once we are convinced that the assembly lines have the intended behaviour, we start with the generation of a production environment. We compile all components to C using the Asf+Sdf compiler (see Section 5). Finally, we combine all the components as discussed in Section 3 resulting in generated tools that make extensive use of rewriting.

Example Applications

The analysis and transformation framework just described can be used in a wide range of applications. In [22] generic analysis functions are used to extract to-be-classes using cluster analysis. In [23] generic analysis support is used to implement type inference for COBOL systems. Rewrite rules using COBOL concrete syntax extract type information about variables occurring in a COBOL system. This information is then used to infer type equivalences, subtype relations, literal elements contained in types, and enumeration types.

As it comes to software renovation factories, we have automated some changes that are needed for correcting the Year 2000 problem. In [54] native patterns are used to remedy a difficult to recognize leap year problem. Another example is the rejuvenation of old systems. In [51] we report on transformations for the restructuring of COBOL/SQL systems. In [17] the control flow of a COBOL/CICS legacy system is normalized. This restructuring improves maintainability, improves performance, and enables connection to the Internet/Intranet using an commercial-off-the-shelf Java to CICS Internet gateway.

Measurements

In our grammar-centric approach, the size of generated tool support is directly related to the size of the grammar. In the case of COBOL, the grammar contains at the moment circa 700 production rules; it includes the embedded languages CICS [31] and SQL [32]. For every production rule we generate at least one generic transformation rule, one analysis rule and native syntax for all access functions for these generic rules. Including these generated rules, we get circa 1100 production rules. The number of generated rewrite rules is in this case approximately 1300.

For the unparser we generate 700 production rules for additional syntax (needed by the unparser) and 700 rewrite rules. Usually, only a few of these generated rules have to be redefined by hand in order to satisfy specific formatting requirements.

The COBOL programs that we transform have at the moment sizes between small (100 LOC) and medium (5–10 KLOC). A transformation that matches 350 times in a 5000 LOC program takes less than a minute to transform.

5 Compilation of ASF+SDF to C

Our second example of a generated tool is the Asf+Sdf compiler itself.

The first goal of the Asf+Sdf compiler is to provide a flexible compilation scheme for the incremental compilation of large modular specifications. The second goal is to generate C code which can process large terms in an efficient way regarding both execution time and memory usage. A third goal is the generation of stand-alone tools (generated C code) that can be shipped to IT industry, without having to ship the Asf+Sdf Meta-Environment as well (see the example of the Risla flattener at the end of this section). We have used our own tools, techniques and formalism while developing the Asf+Sdf compiler. This approach helps achieving the first goal mentioned above, since it makes us the first guinea pigs of our own compiler, The Asf+Sdf specification of the compiler is circa 150 pages (1748 rewrite rules, see Table 1).

There are four aspects of Asf+Sdf that have to be taken into account during compilation:

- Asf+Sdf is a modular algebraic specification formalism. For each module we can define functions and corresponding equations. A complicating factor is the possibility to define a function with some equations in module M and to add extra equations in module N which must of course (in)directly import module M ($M \neq N$).
- The Asf+Sdf formalism provides a default mechanism for the equations, this means that during the rewriting of a term $F(t_1, \ldots, t_n)$ with $n \geq 0$ all non-default equations for function F are tried before the default equations. Note that this is a form of prioritized rewriting [44,3].
- Asf+Sdf provides list matching (also known as associative matching) this mechanism allows a concise style of specification and involves backtracking. Note that associative matching is NP-complete [4] so optimizations for special cases have to be considered in the compiler.
- Asf+Sdf uses innermost rewriting as execution model.

Compilation Strategy

The translation of Asf+Sdf specifications to C code is mainly influenced by the above four aspects of Asf+Sdf. For instance, the fact that Asf+Sdf is based on innermost rewriting is a pleasant convenience because of the call-by-value mechanism of C. This enables us to translate the right-hand side of rewrite rules directly to C function calls. The first aspect (modular structure) on the other hand implies extra work when generating C code. For each Asf+Sdf function a separate C function will be generated. The corresponding equations are translated to conditional statements in this C function. This means that all equations have to be available before the C function can be generated and thus a reshuffling of equations is needed. One consequence of the distribution of equations over several modules is that modules can not serve as incremental compilation units. We have chosen functions as the incremental compilation unit. The C code generated for the default equations should always be executed after the code that is generated for the ordinary equations. List matching patterns are either transformed into ordinary term matching, if the pattern contains no or exactly one list variable, or the patterns are translated to (nested) while-statements in which all elements of the list are inspected until either a successful match is found or no elements are left. See [48] for a general treatment of this topic.

The global compilation strategy is as follows. An Asf+Sdf specification is parsed and for each function with its equations a new Asf+Sdf module is generated containing this function and *all* corresponding equations. We call this flattening of the specification followed by reshuffling of the equations. All constructor functions defined in one module are kept together. These modules are initially used by the compiler for the actual code generation but also when the specification is recompiled to decide for which functions new code should be generated. This phase is entirely independent of the code generation phase. Each new created module is fed to the compiler to generate the actual C code.

The generation of C code is performed in a number of (small) steps. We start with the AsFix representation of each module, essentially the full parse tree of the module still containing layout, comments, keywords and the like. After reshuffling the Asf+Sdf module, it is transformed into another intermediate representation that has been designed to simplify the compilation process. This intermediate formalism μAsf is a simplified prefix representation of Asf+Sdf, see Figure 6 for the μAsf representation of the Booleans. Given this μAsf code, a number of transformations is performed to simplify the actual code generation phase and to increase the performance of the resulting code, e.g., list matching patterns containing at most one list variable are transformed into ordinary term matching. From the resulting transformed μAsf code the C code is generated. The left-hand sides of the μAsf equations are transformed into a matching automaton which is directly included in the C function. The right-hand sides of the equations are directly translated to function calls. See Figure 7 for an example of generated code from the Booleans of Figure 6.

The generated code depends heavily on a general term library (see Section 6.1), which ensures efficient manipulation of terms as well as maximal shar-

```
module Booleans
signature
  true;
  false;
  and(_,_);
  or(_,_);
  not(_);
rules
  and(true,B) = B;
  and(false,B) = false;
  or(true,B) = true;
  or(false,B) = B;
  not(true) = false;
  not(false) = true;
```

Fig. 6. μAsf specification of the Booleans.

```
ATerm and(ATerm arg0, ATerm arg1) {
  if (check_sym(arg0, truesym))
    return arg1;
  if (check_sym(arg0, falsesym))
    return arg0;
  return make_nf2(andsym,arg0,arg1);
}
```

Fig. 7. Generated C code for the `and` function of the Booleans.

ing subterms. The term library provides the basic functionality to construct normal forms, functionality to check the outermost function symbols of terms, and functionality to manipulate the list data structure. Note that the function calls `check_sym` and `make_nf2` in Figure 7 are provided by this library.

Component-based Architecture

The Asf+Sdf compiler has been completely specified in Asf+Sdf and its internal organization is identical to the structure that was already shown in Figure 2. In Figure 8 we show the architecture of the new Asf+Sdf Meta-Environment in which the compiler has been integrated.

The reshuffling of equations already discussed above, is performed by the component `Module-DB` and the C code generation is performed by the component `Compiler` (the actual Asf+Sdf compiler). In the current implementation

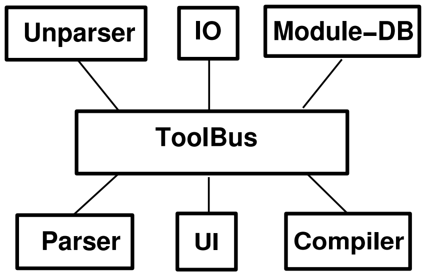

Fig. 8. Simplified architecture of the new Asf+Sdf Meta-Environment.

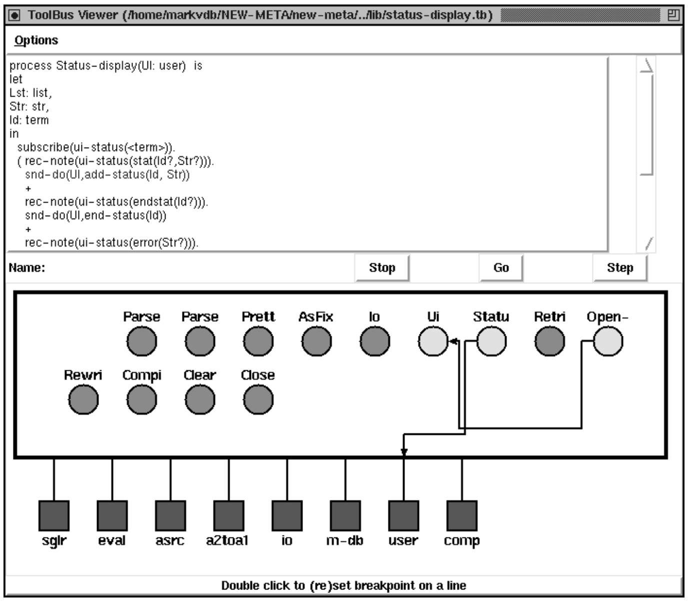

Fig. 9. Run-time view of internal structure of new Asf+Sdf Meta-Environment.

of the new Asf+Sdf Meta-Environment only the compiler and the parser generator are compiled Asf+Sdf specifications. The parser generator is not yet shown in Figure 8. The other components are either implemented in C or Tcl. We stress that some of these are off-the-shelf components.

Run-time view

Figure 9 presents a snapshot of the new Asf+Sdf Meta-Environment [13]. It contains the new Asf+Sdf compiler [14] as a component. This snapshot is obtained via a debug facility connected to the ToolBus: it is possible to view the system at run-time while the architecture is visible. The window at the top in Figure 9 contains a small piece of the ToolBus script describing the coordination between processes and components implementing the new Asf+Sdf Meta-Environment. The window in the middle contains a number of processes coordinating the cooperation of the components. The window at the bottom represents the actual components. The communication between the processes and tools is represented by arrows, in Figure 9 two messages are sent, one from process `Open-module` to process `UI` and one from process `Status` to component `ui`. Note that there is no one-to-one correspondence between processes (we have 13) and components (there are 8), we refer to [7] for more details. Also note that the architecture given earlier (Figure 8) has been slightly simplified in comparison to the actual structure shown in Figure 9.

Measurements

In Table 1 we give some measurements for the compilation of four large Asf+Sdf specifications: compiler (the Asf+Sdf compiler itself), parser gen-

Specification	Nr of eqs	Lines of Asf+Sdf	Lines of C code	Asf+Sdf compiler	C compiler
Compiler	1748	7972	75578	186 s	263 s
Parser generator	1365	4586	49649	203 s	168 s
COBOL formatter	1857	8387	76788	371 s	339 s
Risla flattener	1258	16453	76664	324 s	759 s

Table 1
Measurements of the Asf+Sdf compiler.

erator (a GLR parser generator), COBOL formatter (a generated formatter for the COBOL language), and Risla flattener (a domain-specific language, explained in more detail below).

For each specification we give the number of equations, the number of lines of Asf+Sdf specification without comments, the number of lines of generated C code, the time needed by the Asf+Sdf compiler to generate C code, and finally the time it took the C compiler (using the `-O` optimization option) to compile the generated code. We prefer the native `cc` compiler since it optimizes tail recursion, which is not the case for `gcc` and this yields a better performance in some cases. Measurements were performed on an ULTRA SPARC-IIi (300 MHz) with 576 Mb of memory. We are currently preparing detailed benchmarks comparing the efficiency of Asf+Sdf with that of other, related, formalisms [14].

We now further explain the Risla flattener. Risla is a domain-specific language for specifying financial products developed by Cap Gemini in cooperation with MeesPierson (a Dutch business bank) and CWI and UvA [2,20].

The syntax and semantics of the Risla language have been developed using Asf+Sdf. Several years after the initial design, the language was extended with modular constructs yielding the new language Modular Risla. Since there was already a compiler that generated COBOL from the original Risla language, Modular Risla is translated ("flattened") to original, non-modular, Risla. Both Modular Risla and the flattener have been specified in Asf+Sdf using the Asf+Sdf Meta-Environment. The result is a Modular Risla programming environment.

Unfortunately, flattening large Modular Risla specifications took much time (about 40 minutes CPU time for an average Modular Risla specification and several hours for large ones) because the Asf+Sdf specification defining the flattener is executed by an *interpreter* in the Asf+Sdf Meta-Environment. The new Asf+Sdf compiler was then used to compile the Risla flattener to a stand-alone C program. Cap Gemini incorporated this stand-alone C code within the original Risla compiler. The 40 minutes CPU time mentioned above reduce to about 30 seconds when using the generated stand-alone C code.

6 Research Issues

Before presenting a number of research issues that may interest people working in the field of term rewriting, we give first some further sobering observations on the use of formal techniques in industry,

As already indicated in Section 2, IT industry is wary for new concepts. Some researchers may think that, e.g., correctness proofs will be beneficial for IT industry. This is not true. The return on investment (ROI) is poor: $0.90 after one year $0.80 after two years, $0.70 after three years and $0.60 after four years [35, loc. cit. p. 108]. The ROI of the reusability of high-quality artefacts is excellent: $3.75 after one year $9.15 after two years, $21.75 after three years and $43.75 after four years [35, loc. cit. p. 105]. The list of ROI of selected software technologies contains 143 entries, reuse is number one and correctness proofs are number 136. So if one thinks that we might have an unusual viewpoint on the research issues, namely no termination proofs and no confluence proofs, note that this is confirmed by cost estimations from IT industry (this does not necessarily mean that we do not *like* proofs, see [26] for an example).

In the following subsections we propose therefore research issues that encourage the amalgamation of several theoretical directions. The goal is to achieve high-level, high-quality artefacts that can be reused in a wide range of applications.

6.1 Terms and Term Databases

From a theoretical point of view, terms are a straightforward data type, but in the context of industry-sized applications various non-trivial problems have to be solved:

- What is the most concise representation for terms? For large terms in the mega-byte range, the time needed for file transfers becomes comparable to the time needed for term rewriting.
- Which statistics of term usage are relevant when choosing a term representation? Typically, statistics on the average arity of function symbols and the average length of lists can be used to select term representations that are either optimized for space or for access time.
- How can sharing of subterms be maximized? If maximal sharing is to be implemented—when a new term is about to be created, a lookup has to be performed to check whether that term already exists—what are the best table lookup strategies?
- How can terms be represented in a *language-neutral* form, so that they can be exchanged between tools that are implemented in different languages?
- If we take term rewriting seriously, global information has to be stored in term databases. How do we define and implement efficient storage, retrieval and query operations on such persistent databases?

Currently, we have found reasonable answers to most of the above questions in the design and implementation of the *Annotated Term Format* (ATF), a language-independent prefix format that is supported by efficient implementations in C and Java. However, several issues still need further research. For instance, how do we organize the access of several concurrent tools to a common term database? Initial work on waitfree algorithms to achieve this appears in [30].

6.2 GLR Parsing versus Parallel Term Rewriting

As already observed in Section 4, the use of Generalized LR parsing is essential for solving the parsing problem for large, existing, languages that have many dialects.

LR parsing is based on *shift/reduce* parsing: symbols from the input stream are either "shifted" (i.e., placed on an auxiliary stack) or a "reduce" action is performed (i.e., the top symbols pushed on the stack represent one side of a grammar rule and can be replaced by the other side of the rule). For the class of LR-parsable languages, one can always make a unique choice between shifting or reducing. For the larger class of all context-free grammars this is no longer the case and there may shift/reduce conflicts (reading the next symbol or reducing the current stack) or reduce/reduce conflicts (the stack can be reduced according to different grammar rules).

The basic idea of GLR parsing is to *fork* the parse process at points where several choices are possible. All parses are synchronized on reading the next input symbol and parallel parses are joined whenever possible. In this manner, a parse forest is build that maximizes the sharing between the parallel parses.

A striking similarity exists between GLR parsing and parallel rewriting where a rewrite system may contain critical pairs and parallel reduction takes place for all possible redexes. In principle, the same (parallel) rewriting machinery could be used for both rewriting and GLR parsing.

In the context of the Asf+Sdf Meta-Environment the benefits would be substantial:

- Rather than maintaining *two* forms of rewriting—one for parsing and one for term rewriting—one form would suffice.
- Compilation and optimization techniques have a positive effect on both parsing and term rewriting. Given the increasing efficiency of compiled term rewriting systems, this may have interesting effects on the compilation and optimization of parsers.

6.3 Coordination Languages versus Strategy Languages

One of the great challenges of software engineering is to find effective mechanisms for the evolutionary development and integration of software components. The basic premise is to separate *computation* (the basic operations and calculations to be performed) from *coordination* (the ways is which computations interact and cooperate). It turns out that this separation improves the flexibility and maintainability of software systems.

One example of this approach is the ToolBus coordination architecture, in which a process-oriented calculus is used to describe the cooperation protocol between different tools that may be written in different languages. Essentially, tools perform computation steps and the process script tells which steps are performed when by whom.

If we compare this approach to strategy languages for rewriting there is, again, a striking similarity: the strategy language describes when and where each rewrite rule has to be applied.

7 A matter of Confluence

We have found that questions regarding the confluence of huge rewrite systems are seldomly posed: the specification writer has a mental model of the problem at hand that provides sufficient guidance for confluent execution of the specification as rewrite system. This is fortunate, since the actual determination of the confluence of these huge specifications is hardly feasible. Similar considerations hold for the termination of the term rewriting systems we encounter.

It is, however, another form of confluence that we have argued for in this paper: the confluence of seemingly disparate areas like parallel rewriting versus GLR parsing, and coordination languages versus strategy languages.

Since GLR parsing and coordination languages have turned out to be crucial for implementing industry-sized applications based on term rewriting, we expect that a further exploration and unification of these areas, as suggested in this paper, might be beneficial for both theory and for selling term rewriting to industry.

References

[1] American National Standards Institute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[2] B. R. T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[3] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term-rewriting systems with rule priorities. *Theoretical Computer Science*, 67(2&3):283–301, 1989.

[4] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems.

In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *LNCS*, pages 417–429. Springer-Verlag, 1985.

[5] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[6] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[7] J.A. Bergstra and P. Klint. The discrete time ToolBus—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[8] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer System Science*, 32(3):323–362, 1986.

[9] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of Asf+Sdf. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[10] M.G.J. van den Brand, P. Klint, and P. Olivier. ATerms: Exchanging data between heterogeneous tools for CASL, 1998. Available at ftp://ftp.brics.dk/Projects/CoFI/Notes/T-3/doc.ps.Z.

[11] M.G.J. van den Brand, P. Klint, P. Olivier, and E. Visser. Aterms: representing structured data for exchange between heterogeneous tools, 1996. Unpublished manuscript.

[12] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[13] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Implementation of a prototype for the new Asf+Sdf meta-environment. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer verlag, 1997.

[14] M.G.J. van den Brand, P. Olivier, J. Heering, and P. Klint. Compiling rewrite systems: The Asf+Sdf compiler. Technical report, CWI/University of Amsterdam, 1998. In preparation.

[15] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. Available at http://adam.wins.uva.nl/˜x/scp/scp.html. An extended abstract with the same title appeared earlier: [**?**].

[16] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer verlag, 1997. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[17] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html.

[18] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/~x/ref/ref.html.

[19] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[20] A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39, Erfurt, September 1997. Ilmenau Technical University.

[21] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[22] A. van Deursen and T. Kuipers. Finding classes in legacy code using cluster analysis. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE'97 Workshop on Object-Oriented Reengineering*, Report TUV-1841-97-10. Technical University of Vienna, 1997.

[23] A. van Deursen and L. Moonen. Type inference for COBOL systems. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5th Working Conference on Reverse Engineering*. IEEE Computer Scociety Press, 1998.

[24] A. van Deursen and P.D. Mosses. Executing Action Semantics descriptions using ASF+SDF. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, Workshops in Computing, pages 415–416. Springer-Verlag, 1993.

[25] F. van Dijk, W.J. Fokkink, G.P. Kolk, P. van de Ven, and S.F.M. van Vlijmen. EURIS: a specification method for distributed interlockings (extended abstract). In *Proceedings of SAFECOMP '98*, LNCS. Springer-Verlag, 1998. To appear.

[26] W.J. Fokkink and C. Verhoef. Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories.

Technical Report P9802, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/˜x/cade/cade.ps.

[27] J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn–Kersenboogerd. In *Proceedings of the Tenth Annual Conference on Computer Assurance, Compass'95*, pages 57–68. IEEE, 1995.

[28] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Latest version: ftp://ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

[29] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In The Commission of the European Communities, editor, *Esprit '85 - Status Report of Continuing Work 1*, pages 467–477. North-Holland, 1986.

[30] W.H. Hesselink and J.F. Groote. Waitfree distributed memory management by Create, and Read Until Deletion (CRUD). Technical Report SEN-R9811, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1998.

[31] IBM, Mechanicsburg, Pennsylvania, USA. *CICS/ESA Application Programming Reference*, 1992.

[32] IBM Corporation. *DB2 for MVS/ESA V4 SQL Reference*, 4 release 1 edition, 1997.

[33] V.K. Jolly. *Commercializing New Technologies : Getting from Mind to Market*. Harvard Business School Press, 1997.

[34] Capers Jones. *Assessment and Control of Software Risks*. Prentice-Hall, 1994.

[35] Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, 1998.

[36] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175–193, 1984.

[37] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 129–143. Springer-Verlag, 1988.

[38] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[39] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/˜x/evol-se/evol-se.html.

[40] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[41] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. In H.A. Wijshoff, editor, *Computing Science in the Netherlands (CSN93)*, SION, pages 217–228, 1993.

[42] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[43] J.R. Levine, T. Mason, and D. Brown. Yacc ambiguities and Conflicts. In *lex & yacc*, pages 217–241. O'Reilly & Associates, Inc., 2nd edition, 1992.

[44] C. K. Mohan. Priority rewriting: Semantics, confluence, and conditionals. In N. Dershowitz, editor, *Rewriting Techniques and Applications (RTA '89)*, volume 355 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 1989.

[45] C. K. Mohan and M. K. Srivas. Conditional specifications with inequational assumptions. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems (CTRS '88)*, volume 308 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1988.

[46] C. K. Mohan and M. K. Srivas. Negation with logical variables in conditional rewriting. In N. Dershowitz, editor, *Rewriting Techniques and Applications (RTA '89)*, volume 355 of *Lecture Notes in Computer Science*, pages 292–310. Springer-Verlag, 1989.

[47] R.W. Olshavsky. Time and the rate of adoption of innovations. *Journal of Consumer Research*, 6:425–428, March 1980.

[48] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(5):158–167, 1978.

[49] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

[50] H. Reubenstein, R. Piazza, and S. Roberts. Separating parsing and analysis in reverse engineering tools. In R.C. Waters and C.J. Chikofsky, editors, *Proceedings of the 1st Working Conference on Reverse Engineering*, pages 117–125, 1993.

[51] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintenance. Technical Report P9807, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/˜x/asm/asm.html.

[52] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. Technical Report P9805, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/˜x/cale/cale.html.

[53] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. Technical report, University of Amsterdam, Programming Research Group, 1998. In preparation.

[54] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5th Working Conference on Reverse Engineering*. IEEE Computer Scociety Press, 1998. Available at http://adam.wins.uva.nl/˜x/npl/npl.html.

[55] M. Tomita. *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1986.

[56] Eelco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. Available at
http://www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps.