Bringing Strategic Rewriting into the Mainstream

Emilie Balland, Horatiu Cirstea, Pierre-Etienne Moreau

Bringing Strategic Rewriting into the Mainstream

Emilie Balland[a], Horatiu Cirstea[a,b], Pierre-Etienne Moreau[a,b]

[a] *Inria*
[b] *Université de Lorraine*

Abstract

Developing programming paradigms and languages that ease the programmers' job of writing quality reusable code have been central ever since the beginning of programming. Tom, the language presented here, has been developed in an attempt to make a step forward in this direction. It promotes *term rewriting* techniques piggybacked on top of a general-purpose programming language like Java, C, and Python. This results in concise pieces of code which are close to the targeted application domain and which can be reasoned about using effective theoretical and practical tools. This approach has been validated on a variety of concrete applications ranging from academic tools like theorem provers, to large and complex applications, including the Tom compiler itself and several industrial products. This paper presents an overview of the current version of the language and explains the design rationale, browses the application domains and gives some hints on the tool building.

Keywords: pattern matching, term rewriting, tree traversal, strategies, object-oriented programming

1. Introduction

Since the invention by Charles Babbage of the first general-purpose programmable computing device, computers have required instructions on how to perform specific tasks. For the early analog and digital computers these instructions were specified using relatively limited switch-based mechanical or electrical configurations which had to be manually reconfigured for every new task. Not only that programming these machines was tedious but the resulting programs could not be reused. Following the works of Alan Turing and John von Neumann stored-program electronic machines able to

run different programs without reconfiguration appeared in the early 1950s. Programming in machine specific assembly languages was nevertheless intellectually difficult and error-prone, and high-level languages compiled towards machine code had been subsequently designed for these new computers.

Despite these advances, the formal notations of the general-purpose programming languages made the communication between business experts and software engineers quite burdensome. The first attempt to bridge this gap was probably done with the COBOL language which proposed a syntax closer to natural language and thus easier to grasp by non-programmers. This idea of writing business user readable code is carried through with the development of Domain Specific Languages (DSL), languages which are either specialized for a particular *problem domain* (*e.g.* parsing, database queries, *etc.*) or specialized for a particular *business domain* (*e.g.* mathematics, finance, *etc.*). In addition to code readability, requirements such as safety and performance can be more easily achieved because of the domain restriction.

Tom [1], the language we present here can be classified in the *problem-domain* category since it relies on *term rewriting* techniques to tackle tree-structure manipulation problems. Term Rewriting [2, 3] provides a theoretical framework that has been extensively used to model, to study, and to analyze various parts of a complex system, from algorithms to running software. Term rewriting has also been used as a programming paradigm resulting in several languages such as ASF+SDF [4], ELAN [5], Maude [6], Rascal [7], Stratego [8] and Spoofax [9]. These languages have clear semantics [10, 11], the corresponding programs are often close to the natural description of the targeted domain and effective theoretical and practical tools can be used to check properties on these programs.

The usability of a language depends not only on its expressive power but also on the smooth integration with other software components. In Tom, the expressiveness of rewriting is made available on top of major existing languages which provide general purpose programming features. Since its first version in 2001, the Tom language has been used in various application domains and these applications have greatly influenced the evolution of the language. This paper presents an overview of the current version (2.10) of the Tom language and discusses the design rationale and tool building details that have never been published before.

The paper is organized as follows. In Section 2, we briefly present the current version of the language and the design rationale that has driven

its evolution. The two following sections present respectively representative applications and some key details of the tool implementation. Sections 5 and 6 present related work and conclude.

2. Language Design

The TOM language is a high-level language dedicated to the traversal and transformation of trees, structures largely used in computer science to represent data such as programs, symbolic formulae, parse trees, abstract syntax trees (AST), XML documents, *etc.* The main advantages of using such a language are the programs which are concise and close to the respective application domain, and the possibility to reason about these specific pieces of code. TOM relies largely on the concepts of rewrite rules and strategies and its design has been strongly influenced by more than 20 years of research in this area, particularly by ELAN, ASF+SDF, Stratego, MAUDE, and JJTraveler. When developing ELAN, one of the most important lessons we learned is that *integration capabilities* are essential to make such research tools widely used both in academic and industrial projects. From this perspective, when developing TOM, the following requirements have served as design guidelines:

1. *Integration into mainstream programming environments.* Using TOM in a development team or in an industrial context should have almost no impact on the corresponding development process and thus minimize the learning curve.
2. *Accommodate various domains and applications.* TOM programs should be able to cover a broad range of domains and should be integrated easily into existing applications.
3. *Zero-overhead principle.* "What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better" [12].

To fulfill the first design requirement, the TOM language is piggybacked on top of a *host* programming language and preserves thus all available supporting components (*e.g.* input/output, graphical user interface, database connectivity, *etc.*). The main interest of this piggybacking approach is to bring high-level domain-specific concepts into confirmed programming languages, while leveraging integrated-development environments and APIs of the underlying general-purpose language. Notorious examples of such an approach are LEX and YACC, where scanning and parsing capabilities are

added to the C language. The TOM constructs are general enough to support a broad range of underlying programming languages, namely JAVA, C, C++, Python, OCaml and ADA. For presentation purposes, we consider in the following that JAVA is the underlying host language.

TOM can be used to manipulate data structures built using its specific data representation as well as any kind of data structure defined in the host language just by providing a *mapping* which defines the correspondence between the two representations. This feature contributes to the second design requirement since TOM can be used for implementing or refactoring functionalities of an existing application without requiring a new tree representation for the objects being handled. For example, in the domain of model-driven engineering, TOM is directly usable with standard model representations such as the Eclipse Modelling Framework [13] (EMF) ones.

To fulfill the last design requirement, we kept the language as small as possible with every new TOM construct providing significant added value and generating no overhead in terms of efficiency and maintenance costs. The problem domain of TOM concerns three main areas: (1) the representation of the domain model using *algebraic signatures*, (2) the decomposition of complex data manipulations into elementary steps, encoded by *rewrite rules* and (3) the control of rule application using *strategies*. Each of these steps corresponds to a family of language construct which can be used independently. In the rest of this section, we briefly present these main constructs; for a detailed presentation of the TOM language, the reader may refer to [14, 15].

2.1. Representing the domain model

When developing an application in TOM the domain model is represented by tree-shaped data-structures defined using algebraic signatures. Informally, a signature is a list of *sorts* associated with a list of *constructors* described by their name and their domain. For instance, the notion of arithmetic expression can be described using the following `%gom` construct:

```
%gom {
  module SymbolicExpression
  imports int String
  abstract syntax
    Expression = Plus(e1:Expression,e2:Expression)
               | Mult(e1:Expression,e2:Expression)
               | Var(name:String)
```

```
                | Cst(n:int)
}
```

In this example, the module `SymbolicExpression` defines the sort `Expression` whose constructors are `Plus`, `Mult`, `Var` and `Cst`. For each constructor we should specify the names of its arguments and their types. A module may import other modules including predefined modules like `int` and `String`.

Tree-shaped objects can be built in the host language using the "`‘`" construct. For example, the TOM+JAVA code fragment

```
Expression t_3_2 = ‘Plus(Cst(3),Cst(2));
```

can be used to build the tree representation of the expression "$3+2$" and to store it in a variable of the appropriate type. The "`‘`" construct can be used anywhere a JAVA expression can be used and its lexical scope corresponds to a well-formed term. In the above example, it ends just before the "`;`".

As mentioned previously, if a data model has already been specified in JAVA then, the corresponding objects can still be manipulated in TOM. This is possible due to the TOM provided *mapping formalism* which can be used to describe the relationship between the concrete implementation and the algebraic signature [1, 15]. A mapping is a piece of code that gives TOM the information about the algebraic structure of the objects we intend to manipulate and, in particular, specifies how to test the algebraic type of the object and how to decompose it. This idea, related to P. Wadler's views [16], allows TOM to match any kind of data structure, as long as a mapping is provided. This *object-tree mapping* is similar to the *object-relational mapping* techniques [17] but the conversion is done towards a tree data-structure instead of a relational database. When using the `%gom` construct, the compiler generates a set of JAVA classes which provide an implementation for the algebraic signature together with a *mapping* making the data structure immediately usable in a TOM program.

2.2. Defining data manipulation

Once the data model defined, data manipulation is specified using rewrite rules declared using the `%match` construct. This construct provides pattern matching capabilities, *i.e.* rules of the form `pattern -> rhs` similar to the ones that exist in functional programming languages. The `pattern` is built upon the algebraic signature that describes the structure of the objects being matched and the right-hand side (`rhs`) of the rule is a list of instructions

written in TOM+JAVA. This code, also called the *action* of the rule, is executed each time the pattern matches.

The following JAVA method uses the `%match` construct to print additions and multiplications whose arguments are constants:

```
public static void print(Expression exp) {
  %match(exp) {
    Plus(Cst(x),Cst(y)) -> { System.out.println( `x + `y ); }
    Mult(Cst(x),Cst(y)) -> { System.out.println( `x * `y ); }
  }
}
```

If this method is called with the previously defined variable `t_3_2` as argument then, the first pattern will match it, and the variables `x` and `y` will be respectively instantiated by 3 and 2, resulting in the printing of 5. Note that variables such as `x` or `y` do not need to be declared: they are local to each rule and their types are automatically inferred. The second rule does not fire since the pattern does not match.

For JAVA developers, the semantics of `%match` should be quite intuitive since, as for the `switch/case` construct of JAVA, the action part is executed for all the patterns that match the subject. The patterns are evaluated from top to bottom and statements such as "`break`" or "`return`" may of course be used to interrupt the execution control flow.

In addition to standard matching, TOM provides a more powerful form of matching called *associative matching with neutral element* [2]. This leads to an enhanced expressive power when searching for elements in an array or a list data structure. For instance, such a matching is particularly well suited for implementing `String` and XML transformations.

2.3. Controlling rule application

Rewrite rules should be generally controlled and combined in order to accomplish the tasks the corresponding application is designed for. Take for example the simple arithmetic rules $0 + x \to x$ and $x + 0 \to x$ which say that adding 0 to an expression has no effect on the respective expression. These rules can be easily implemented in TOM, or in any functional language, but some extra function calls have to be added to encode the recursive simplification of an expression, or the repetitive application until an irreducible form is reached. Nevertheless, mixing rules and their control sometimes makes

difficult the reuse of rules in another context, where rules have to be applied in a different order, for instance.

Rewriting based languages provide more abstract ways to express the control of rule applications through the concept of *strategy*. Like its ascendants, TOM provides a flexible and expressive strategy language allowing the definition of high-level strategies obtained by combining low-level primitives. Among the most important low-level operators we can mention `Sequence(s1,s2)` (which sequentially applies `s1` and `s2` like the Unix pipe operator), and `All(s)` (which applies `s` to all immediate sub-terms of a given term). These basic operators are combined to define high-level strategies such as the bottom-up strategy `BottomUp(s)` $\triangleq$ `Sequence(All(BottomUp(s)),s)`, which applies the strategy given as argument to all the sub-terms of a term in a *bottom-up* manner. Strategies such as *top-down*, *repeat*, or *leftmost-innermost* can be defined similarly.

Besides the elementary strategies `Identity` (which always succeeds), and `Fail` (which never succeeds), in TOM, a set of *transformation rules* (possibly reduced to a singleton) is also a strategy. For example, the above evaluation rules can be expressed as the strategy:

```
%strategy Eval() extends Identity() {
  visit Expression {
    Plus(Cst(0),x) -> { return `x; }
    Plus(x,Cst(0)) -> { return `x; }
} }
```

The `%strategy` construct is used here to declare a set of rules whose name is `Eval`. The `extends Identity()` construct specifies that by default (*i.e.* when no rule matches) the strategy behaves as the identity function. Its application to the term `t_0_3 = `Plus(Cst(0),Cst(3))` can be computed by `` `Eval().visit(t_0_3) ``, resulting in the new term `Cst(3)`. When applied to the term `t_0_3_0 = `Plus(Cst(0),Plus(Cst(3),Cst(0)))` only one rewrite step is performed and the resulting term `` `Plus(Cst(3),Cst(0)) `` is obtained. This latter term could be further reduced by applying `Eval` a second time but a more natural approach consists in applying the bottom-up strategy `` `BottomUp(Eval()).visit(t_0_3_0) `` which results immediately in the term `Cst(3)`. In a complete implementation of arithmetic evaluation, the strategy `Eval` would obviously contain the implementation of all the mathematical simplification rules and a fixed-point (*e.g.* innermost) evaluation strategy would be used.

3. Application Domains

Tom has been used to implement many applications ranging from academic tools like theorem provers, to large and complex applications, including the Tom compiler itself and several industrial products. It has also been used as a teaching support tool not only to illustrate and practice the rewriting concepts but also to rapidly prototype compilers, operational semantics, model checkers, and other notions involving inference and transformation rules. In this section, we focus on some representative application domains and briefly discuss some impacts on the design of the Tom language itself.

3.1. Program transformation

The Tom language has been used for various applications requiring program transformations, including the Tom compiler itself. These transformations are performed at the AST level but Tom can be easily interconnected with the popular ANTLR parser generator when used in a compiler development context.

Tom *compiler.* Strategies and rules are intensively used[1] in the Tom compiler, particularly in the optimization phase which modifies the generated code to make it more efficient. For example, the following rule transforms a sequence of incompatible conditional statements (*i.e.* whose tests could not be evaluated to true in the same time) into a more efficient and semantically equivalent nested conditional statement:

```
%strategy InterBlock() extends Identity() {
  visit Instruction {
    concInstruction(X1*,  If(c1,then1,else1),
                          If(c2,then2,Nop()),  X2*) -> {
      if(incompatible(‘c1,‘c2)) {
        return ‘concInstruction(X1*,If(c1,then1,
              concInstruction(else1,If(c2,then2,Nop()))),X2*)
} } } }
```

The optimizer contains approximately 15 semantics preserving rules (*i.e.* optimizations do not change the behavior of programs) which have been designed independently from any application strategy. The complete separation

[1] approximately 2000 rules and 150 strategies.

between the optimization rules and the application strategy used to combine them allows a strategy independent correctness proof of this optimization phase and an efficient and extensible implementation. The design of the Tom compiler is detailed in Section 4.

Java *bytecode manipulation.* Another application of Tom in the domain of program transformation is the *on-the-fly* strategy compiler presented in [18]. This compilation method based on bytecode specialization is expressed with Tom rules and strategies and performs just-in-time optimizations on the bytecode of compiled Tom strategies.

Bytecode mappings and subsequent bytecode rewriting have also been used for secure class loading. More precisely, defensive class loaders that redirect method invocations to new targets (*e.g.* safe IO API) are implemented by rewriting bytecode just before class loading. This application relies on elaborate strategies to check conditions on the control flow of the corresponding bytecode.

Translation of database queries (from MDX to SQL). Tom has been used by BusinessObjects/SAP to implement on-the-fly translations from MDX queries (Multidimensional Expressions) to SQL queries. The high-level description using rewriting rules and strategies as well as their efficient implementation was determinant in the choice of Tom. The main benefit of this approach used in the Crystal Reports software is its extensibility combined with a better confidence in the implementation due to code readability.

3.2. XML handling

Whatever its concrete internal representation is, any XML document can be seen as a tree. Thanks to a mapping defined for the DOM representation offered by the `w3c.dom` package, complex XML transformations can be accomplished by Tom strategic rewriting. Moreover, Tom has been extended with concrete XML syntax constructs [19] such that both patterns and backquote terms can be written using a mix of XML and plain term-based syntax. For example, the following elementary strategy can be used to simplify an XML document by specifying that all `Person` nodes with an attribute `sex` equal to `"M"` are replaced by `Male` nodes:

```
%strategy printOwner() extends Identity() {
  visit TNode {
   <Person sex="M">x</Person> -> { return `xml(<Male>x</Male>)}
```

```
} }
```

The algebraic sort `TNode` corresponds to the `Node` class of `w3c.dom`.

Comparing to approaches like XSLT, the proposed pattern matching on XML structures is semantically well defined and efficient, thanks to a compiled approach. Additionally, the integration into Java is beneficial when complex computations are needed in addition to XML manipulations. This extension has been used in several industrial products and in particular by Cril Technology to interconnect model checkers for timed automata [20].

3.3. Model-driven engineering

As explained in [21], several architectural approaches have been proposed for defining model transformations: (1) direct model manipulation, for instance, in Java using EMF (Eclipse Modelling Framework [13]); (2) intermediate representation manipulation, for instance, using XSLT as a transformation engine for XML; and (3) high-level model manipulation based on dedicated language support such as ATL [22] or QVT [23]. By providing a mapping for EMF generated data-structures, Tom can be seen as an intermediate approach between (1) and (3) to define model transformations. In complement to pattern matching and rewrite rules capabilities, Tom provides a tool (`emf-generate-mappings`) that automatically generates mappings for EMF meta models [24]. Recently, some new constructs have been added to Tom to automatically generate, during a model transformation process, the meta-models and models that record the relationship between source elements and target elements. These extensions have been used in a project involving Airbus, to both define transformations of AADL (Avionics Architecture Description Language) components implemented in EMF models and to provide traces that can be exploited to verify properties on the generated model w.r.t. the input model.

3.4. Specification and verification

An application domain of predilection for term rewriting in general and for Tom in particular is the area of formal methods. Indeed, rewriting has been used in semantics in order to describe the meaning of programming languages [25] but also to perform deduction when describing by inference rules a logic [26], a theorem prover [27] or a constraint solver [28]. In particular, Tom's associative matching and traversal strategies have been intensively

used to implement inference engines, operational semantics interpreters, and model checking algorithms in an elegant and efficient way.

Lemuridae[2], a proof assistant for superdeduction [29] (a dynamic extension of sequent calculus), is written in Tom. The expressiveness of Tom patterns makes the kernel proof-checker only one hundred lines long leading thus to a high degree of confidence in the prover. Tom was also used for developing an interpreter for the *rho-calculus with explicit substitutions* [11]. The interpreter exploits all the capabilities of Tom resulting in a short implementation close to the operational semantics of the calculus. The calculus itself is expressed using rewrite rules and parametrized strategies, while the interactive features and user interface operations take advantage of the underlying Java language. An automatic prover for the *system BV of the calculus of structures* [30] has been also developed in Tom. In this case, clever normalisation strategies allow an efficient representation of the associative-commutative with neutral element operators while strategy constructs allow the management of the high level of non determinism introduced by deep inference. Tom has also been used by the start-up H&S Information Systems to implement the many structural transformations required in their automated theorem prover and disjunctive logic solver system.

4. Tool Implementation

The Tom system is a unified environment that offers a compiler for the language, vim/emacs styles for editing, ant plugins, an Eclipse[3] plugin, and several tools to easily connect data structures to parsers such as ANTLR and frameworks such as EMF. While being a stable system available for industrial and academic usage, Tom is also a research laboratory to experiment new ideas and new language constructs.

The Tom system can be easily extended with new language features while preserving tool stability. We have organized the system in several components (algebraic signature, pattern matching, strategies) that can be used either independently or in a combined way. For example, in 2004 we provided Gom, a generator of maximally shared and typed data-structures. In 2006 we provided Sl, a strategy library to support the control of rules and the traversal of data structures. These two tools can be used either in a pure

[2] http://rho.loria.fr/lemuridae.html

[3] http://www.eclipse.org/

JAVA application or take advantage of the TOM language constructs. Each component addresses a very precise problem domain, enforcing separation of concerns and easing evolution.

Since its first version in 2001, TOM itself has been written using TOM. As shown in Figure 1, the compiler is organized in a pure functional style as a pipeline of program transformations (type inference, compilation, optimization, generation). At the implementation level we use a plugin architecture in which each compilation stage is implemented by a plugin that transforms a non-mutable abstract syntax tree (implemented by a GOM data structure) using rewrite rules and strategies. The result of the overall transformation is an AST representing a pure JAVA program. With this approach we can easily build different versions of TOM, experimenting and eventually integrating new stages such as the new type inference engine recently introduced.

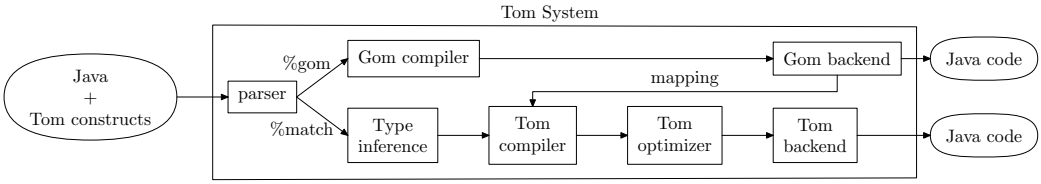

Figure 1: General architecture of the TOM system

The complete environment has been integrated into Eclipse providing a simple and efficient user interface to develop, compile, and debug TOM applications.

From a team management point of view we organized the research and development in such a way that the software is not the property of individuals, but the property of the team. We apply agile method development techniques using, in particular, pair-programming, and any member of the team is able to modify any piece of code. As the compiler is developed in TOM+JAVA, we are able to use standard development tools such as JUnit for unitary testing or Hudson for continuous integration. One key point is that the software should be operational at any time[4].

The combination of these two approaches, decomposing the system in small pieces of independent software and being able to modify any piece

[4]All these nice ideas have been strongly inspired by our exchanges with researchers from CWI and in particular Paul Klint, Mark van den Brand and Jurgen Vinju.

of code written by another member of the team, made us very effective to maintain the system, fix bugs, and experiment new features.

Compiling the compiler. The sources of the TOM compiler are organized into two main directories:

- `stable` contains a version of the compiler fully written in JAVA. This corresponds to the files generated by the bootstrapping process. It can be compiled using the `build.sh` command.
- `src` contains the sources of the compiler, but written in TOM+JAVA. They can be compiled using the `build.sh src` command. More than 12,000 lines of unit tests are used to check the freshly built compiler.

The distribution also contains numerous examples (50,000 lines of code) that are used to illustrate the various constructs of the language. They are also used to check that no regression has been introduced into the system. This comprehensive test suite takes approximately 45 minutes to execute. From our experience, the development of such large test suites is essential to ensure high quality software, especially in an academic context. This intensive testing combined with a bootstrapping process make that in more than 10 years of development we had very few bugs that were present in released versions.

Below, we give statistics about the size of the source code. We can observe that the software is rather compact (approximately 41,000 lines of source code) and this is, in particular, due to the language expressiveness. Indeed, the generated code is considerably larger and corresponds to more than 3,000 classes and 150,000 lines of JAVA code.

	number of lines		number of files
sources (TOM+JAVA)	32,552	31%	90
sources (pure JAVA)	8,715	8%	73
tests	12,471	12%	93
examples	50,916	49%	299
total	104,654	100%	555

Distribution and Installation. The software is available in *binary* and *source* versions and they can both be downloaded via the Inria gforge server[5]. The

[5] `http://gforge.inria.fr/projects/tom/`

current version of the sources can also be obtained via an anonymous `git` access[6]. From a user perspective, the Tom system is easy to install[7] and an extensive documentation is available on the Tom web site[8].

Joined with this paper, we provide both a binary and a source version of the system, including the test suite and the examples.

5. Related Work

Comparing to other term rewriting based languages, like ASF+SDF [4], ELAN [5], Maude [6], Rascal [7] and Stratego [8], the main benefit of Tom is its embedding in Java, easing its integration with existing software components. There exist other languages providing pattern-matching extensions for Java such as Pizza [31], Scala [32], and JMatch [33] but these languages only offer basic pattern-matching, excluding list-matching and strategy capabilities for example.

Regarding data-structure definition, the implementation of the Gom generator has been done in strong cooperation with the authors of ApiGen and followed our experience with the ATerm [34] library used by ASF+SDF. The originality of our solution is to provide typed constructs resulting in a faster and safer implementation.

As far as it concerns the strategies, the design of the Sl library has been greatly inspired by the term rewriting based languages ELAN and Stratego. Comparing to ELAN, Sl does not support implicit non-deterministic strategies, implemented using back-tracking. Instead, the evaluation context is made explicit, allowing one to encode non-deterministic strategies. Comparing to Stratego, the original features of Sl are the reflexivity and the explicit evaluation context capabilities. In object-oriented languages, the work closest to ours are JJTraveler [35] and Kiama [36]. JJTraveler is a framework providing generic visitor combinators for Java; the main difference with the Tom strategy language lies in the lack of explicit recursion and reflexivity. The Kiama library is a domain-specific language embedded in Scala and dedicated to language processing. The strategy language provided by Kiama is very similar to Stratego; the main benefit of Kiama is its integration with Scala, providing an embedded strategy language with a syntax very similar

[6] `git clone https://gforge.inria.fr/git/tom/tom.git`

[7] `http://tom.loria.fr/wiki/index.php5/Documentation:Installation`

[8] `http://tom.loria.fr`

to Stratego's one. Contrary to Tom, Kiama is interpreted and thus some optimizations such as pattern-matching optimizations cannot be realized. For a more detailed comparison, the reader can refer to [37].

6. Conclusion

In this paper, we have given a tour of Tom, a domain-specific language dedicated to tree-structure manipulation. Tom is piggybacked on top of general-purpose programming languages leading to a smooth integration into mainstream development environments. It has been used in various application domains ranging from complete academic theorem provers developed from scratch to software components integrated in industrial products like Crystal Reports.

The Tom compiler is a mature software but with a highly modular architecture allowing a straightforward integration of new features and components. The compiler and the libraries are accompanied by a large number of examples and tests which are used not only to illustrate the main feature of Tom but also to ensure a high level of confidence by extensive non-regression tests. The system comes also with a detailed documentation and with extensive tutorials and teaching materials.

Several extensions and applications are currently under consideration and development. We can mention property based testing tools inspired by quickcheck [38], intended to enforce the confidence of corresponding Tom tested programs, and the application of rewriting strategies to the automatic synthesis of proofs for multi-scale systems [39].

References

[1] P.-E. Moreau, C. Ringeissen, M. Vittek, A pattern matching compiler for multiple target languages, in Proceedings of CC 2003, Vol. 2622 of LNCS, Springer, 2003, pp. 61–76.

[2] F. Baader, T. Nipkow, Term *Re*writing and *all That*, Cambridge University Press, 1998.

[3] Terese, Term Rewriting Systems, Cambridge University Press, 2003, M. Bezem, J. W. Klop and R. de Vrijer, eds.

[4] M. van den Brand, al., The ASF+SDF Meta-Environment: a Component-Based Language Development Environment, in Proceedings of CC 2001, Vol. 2027 of LNCS, Springer, 2001, pp. 365–370.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, An overview of ELAN, in Proceedings of WRLA 1998, Vol. 15 of ENTCS, Elsevier, 1998.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, The Maude 2.0 System, in Proceedings of RTA 2003, Vol. 2706 of LNCS, Springer-Verlag, 2003, pp. 76–87.

[7] P. Klint, T. van der Storm, J. J. Vinju, RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation, in Proceedings of SCAM 2009, IEEE Computer Society, 2009, pp. 168–177.

[8] E. Visser, Z.-e.-A. Benaissa, A. Tolmach, Building program optimizers with rewriting strategies, in Proceedings of ICFP 1998, ACM Press, 1998, pp. 13–26.

[9] L. C. L. Kats, E. Visser, The Spoofax language workbench. Rules for declarative specification of languages and IDEs, in Proceedings of OOPSLA 2010, 2010, pp. 444–463.

[10] N. Martí-Oliet, J. Meseguer, Rewriting logic as a logical and semantic framework, in Proceedings of WRLA 2000, Vol. 4 of ENTCS, Elsevier, 2000.

[11] H. Cirstea, C. Kirchner, The rewriting calculus — Part I *and* II, Logic Journal of the Interest Group in Pure and Applied Logics 9 (3) (2001) 427–498.

[12] B. Stroustrup, The design and evolution of C++, Pearson Education India, 1994.

[13] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework 2.0, Addison-Wesley Professional, 2009.

[14] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, A. Reilles, Tom: Piggybacking rewriting on Java, in Proceedings of RTA 2007, Vol. 4533 of LNCS, Springer, 2007, pp. 36–47.

[15] Documentation of Tom 2.10, `http://tom.loria.fr` (2013).

[16] P. Wadler, Views: a way for pattern matching to cohabit with data abstraction, in Proceedings of POPL 1987, ACM Press, 1987, pp. 307–313.

[17] S. Melnik, A. Adya, P. A. Bernstein, Compiling mappings to bridge applications and databases, in Proceedings of the SIGMOD 2007, ACM, 2007, pp. 461–472.

[18] E. Balland, P.-E. Moreau, A. Reilles, Bytecode rewriting in Tom, in Proceedings of BYTECODE 2007, Vol. 190, Elsevier, 2007, pp. 19–33.

[19] H. Cirstea, P.-E. Moreau, A. Reilles, TomML: A rule language for structured data, in Proceedings of RuleML 2009, Vol. 5858 of LNCS, Springer, 2009, pp. 262–271.

[20] B. Tavernier, Calife: A generic graphical user interface for automata tools, in Proceedings of LDTA 2004, Vol. 110 of ENTCS, Elsevier, 2004, pp. 169 – 172.

[21] S. Sendall, W. Kozaczynski, Model transformation: the heart and soul of model-driven software development, Software 20 (5) (2003) 42–45.

[22] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of Computer Programming 72 (1-2) (2008) 31–39.

[23] Object Management Group, Inc., Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0 (2008).

[24] J.-C. Bach, X. Crégut, P.-E. Moreau, M. Pantel, Model transformations with tom, in Proceedings of LDTA 2012, ACM, 2012.

[25] G. Kahn, Natural semantics, Tech. Rep. 601, INRIA Sophia-Antipolis (Feb. 1987).

[26] J.-Y. Girard, Y. Lafont, P. Taylor, Proofs and Types, Vol. 7 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[27] J.-P. Jouannaud, H. Kirchner, Completion of a set of rules modulo a set of equations, SIAM Journal of Computing 15 (4) (1986) 1155–1194.

[28] J.-P. Jouannaud, C. Kirchner, Solving equations in abstract algebras: a rule-based survey of unification, in Computational Logic. Essays in honor of Alan Robinson, The MIT press, 1991, Ch. 8, pp. 257–321.

[29] P. Brauner, C. Houtmann, C. Kirchner, Principles of superdeduction, in Proceedings of LICS 2007, IEEE Computer Society, 2007, pp. 41–50.

[30] A. Reilles, Canonical abstract syntax trees, in Proceedings of WRLA 2006, Vol. 176, ENTCS, 2006, pp. 165–179.

[31] M. Odersky, P. Wadler, Pizza into Java: Translating theory into practice, in Proceedings of POPL 1997, ACM Press, 1997, pp. 146–159.

[32] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An overview of the Scala programming language, Tech. rep., EPFL (2004).

[33] J. Liu, A. C. Myers, JMatch: Iterable abstract pattern matching for java, in Proceedings of PADL 2003, Vol. 2562 of LNCS, Springer, 2003, pp. 110–127.

[34] M. van den Brand, H. de Jong, P. Klint, P. Olivier, Efficient annotated terms, Software, Practice and Experience 30 (3) (2000) 259–291.

[35] J. Visser, Visitor combination and traversal control, in Proceedings of OOPSLA 2001, ACM Press, 2001, pp. 270–282.

[36] A. Sloane, Lightweight language processing in Kiama, in Proceedings of GTTSE 2011, Vol. 6491 of LNCS, Springer, 2011, pp. 408–425.

[37] E. Balland, P.-E. Moreau, A. Reilles, Effective strategic programming for java developers, Software: Practice and Experience 44 (2) (2014) 129–162.

[38] K. Claessen, J. Hughes, Quickcheck: A lightweight tool for random testing of haskell programs, in Proceedings of ICFP 2000, ACM, 2000, pp. 268–279.

[39] W. Belkhir, A. Giorgetti, M. Lenczner, A symbolic transformation language and its application to a multiscale method, Journal of Symbolic Computation (2014)