# TomML: A Rule Language For Structured Data

Horatiu Cirstea, Pierre-Etienne Moreau, and Antoine Reilles

Université Nancy 2 & INRIA & LORIA
BP 239, F-54506 Vandoeuvre-lès-Nancy, France
first.last@loria.fr

**Abstract.** We present the Tom language that extends Java with the purpose of providing high level constructs inspired by the rewriting community. Tom bridges thus the gap between a general purpose language and high level specifications based on rewriting. This approach was motivated by the promotion of rule based techniques and their integration in large scale applications. Powerful matching capabilities along with a rich strategy language are among Tom's strong features that make it easy to use and competitive with respect to other rule based languages. Tom is thus a natural choice for querying and transforming structured data and in particular XML documents [1]. We present here its main XML oriented features and illustrate its use on several examples.

## 1 Introduction

Pattern matching is a widely spread concept both in the computer science community and in everyday life. Whenever we search for something, we build a so-called pattern which is a structured object that specifies the features we are interested in. Mathematics makes a full use of patterns, some quite elaborated, and this is similar in logic and computer science.

The complexity of the matching process obviously depends on the complexity of the objects it targets. Matching a shape in a picture is significantly more difficult than recognizing a word in a text. There is also a compromise between the complexity of the data and the facility to manipulate it. The eXtensible Markup Language (XML) is a specification language that proved to be an excellent option for describing data since it allows the description of complex structures in a computer friendly format well suited for matching and transformation. In particular, XML data is organized in a tree structure having a single root element at the top and this is exactly the kind of objects manipulated in classical pattern matching algorithms and corresponding tools.

Of course, matching is generally not a goal by itself: if matching is done, this is because we want to perform an action. Typically, if we have detected that zero is added to some number, we would like to simplify the expression, for example by replacing $7 + 0$ by 7. Similarly, XML documents are often transformed into HTML documents for a better presentation in web pages. This transformation can be naturally described using rewriting that consists intuitively in matching an object and (partially) replacing it by another one.

The rewriting concept appears from the very theoretical settings to the very practical implementations. Several rewriting languages, like ASF+SDF [2], Elan [3] and Maude [4], have been developed on top of efficient pattern-matching algorithms and use sophisticated techniques for optimizing the (strategic) rewriting computations. Nevertheless, the libraries and facilities (*e.g.* input/output, threads, interfaces, etc.) are relatively limited compared to largely used languages like Java, for example. The language Tom (`tom.loria.fr`) implements the concept of *Formal Island* [6] that consists in making a specific technology available on top of an existing language. In the case of Tom [5] this technology is the strategic term rewriting and the existing language is Java (although the connection is also possible with potentially any other language).

The Tom programs have a clear semantics based on term rewriting and thus can be subject to formal analysis using the tools available in the domain and, on the other hand, a great potential for cross-platform integration and usability. We recover the portability and re-usability features of XML and Tom looks thus a natural choice when one wants to query and transform XML documents.

We present here an extension of Tom, called TomML, that makes available all the Tom features into a syntax adapted to XML document manipulation. In fact, this extension can be easily adapted to accommodate any other structured data provided that a tree representation can be obtained out of it.

We briefly introduce Tom in the next section. In Section 3 we illustrate via several examples the main pattern-matching features of TomML and in Section 4 we show how strategies can be used to give more expressive power to the formalism. We finish with a brief comparison with similar tools and some concluding remarks.

## 2 Tom in a nutshell

As we have already said, Tom is an extension of Java which adds support for algebraic data-types and pattern matching. One of the most important constructs of the language is `%match`, a pattern matching construct which is parametrized by a list of objects, and contains a list of rules. The left-hand sides of the rules are pattern matching conditions (built upon Java class names and variables), and the right-hand sides are Java statements. Like standard `switch/case` construct, patterns are evaluated from top to bottom, firing each action (*i.e.* right-hand side) whose corresponding left-hand side *matches* the objects given as arguments.

For instance, assume that we have a hierarchy of classes composed of `Account` from which inherit `CCAccount` (credit card account) and `SAccount` (savings account), each with a field `owner` of type `Owner`. Given two objects `s1` and `s2`, the following code prints the owner's name if it is the same for the two accounts and if these accounts are of type `CCAccount`, respectively `SAccount`, and just prints the text `"CCAccount"` if both accounts are of type `CCAccount`:

```
%match(s1,s2) {
  CCAccount(Owner(name)),SAccount(Owner(name)) -> { print(name); }
  CCAccount(_),CCAccount(_) -> { print("CCAccount"); }
}
```

In the above example, `name` is a variable. Notice the use of the *non-linearity* (*i.e.* the presence of the same variable at least twice in the pattern) to denote concisely that the same value is expected. The “_” is an anonymous variable that stands for anything. The equivalent Java code would be:

```
if (s1 instanceof CCAccount) {
  if (s2 instanceof SAccount) {
    Owner o1=((CCAccount)s1).getOwner();
    Owner o2=((SAccount)s2).getOwner();
    if (o1 != null && o2 != null) {
      if ((o1.getName()).equals(o2.getName())) {
        print(o1.getName());
      }
    }
  } else { if (s2 instanceof CCAccount) { print("CCAccount"); } }
}
```

Besides matching simple objects, Tom can also match lists of objects. For instance, given a list of accounts (`List<Account> list`), the following code prints all the names of the credit card accounts' owners:

```
%match(list) {
  AccountList(X*,CCAccount(Owner(name)),Y*) -> { print(name); }
}
```

`AccountList` is a variadic list operator, the variables suffixed by `*` are instantiated with lists (possibly empty), and can be used in the action part: here `X*` is instantiated with the beginning of the list up to the matched object, whereas `Y*` contains the tail. The action is executed for each pattern that matches the subject (assigning different values to variables). Patterns can be non-linear: `AccountList(X*,X*)` denotes a list composed of two identical sublists, whereas `AccountList(X*,x,Y*,x,Z*)` denotes a list containing two occurrences of one of its elements. Another feature of Tom patterns that is worth mentioning is the possibility to embed negative conditions using the complement symbol “!” [7]. For instance, a so-called *anti-pattern* of the form `!AccountList(X*,CCAccount(_),Y*)` denotes a list of accounts that does not contain a credit card account. Similarly, `!AccountList(X*,x,Y*,x,Z*)` stands for a list containing only distinct elements, and `AccountList(X*,x,Y*,!x,Z*)` for one that has at least two distinct elements. There is no restriction on patterns, including complex nested list operators combined with negations. This allows the expression of different algorithms in a very concise and safe manner.

Since its first version in 2001, Tom itself has been written using Tom. The system is composed of a compiler and a library which offers support for predefined data-types such as integers, strings, collections, and many other Java data-structures. The compiler is organized, in a pure functional style, as a pipeline of program transformations (type inference, simplification, compilation, optimization, generation). Each phase transforms a Java+Tom abstract syntax tree using rewrite rules and strategies. At the end a pure Java program is obtained.

The complete environment is integrated into Eclipse (`www.eclipse.org`) providing a simple and efficient user interface to develop, compile, and debug rule based applications. It has been used in an industrial context to implement several large and complex applications, among them a query optimizer for Xquery and a platform for transforming and analysing timed automata using XML manipulation. On several classical benchmarks Tom is competitive with state of the art rule-based implementations and functional languages.

## 3 TomML, an extension for XML manipulation

One of the main objectives of Tom is to be as generic as possible. The implementation of the handled data-structures is not hard-wired in the system but becomes a parameter of the compiler. For that, we have introduced the notion of *formal anchor*, also called *mapping*, which describes how a concrete data-structure (*i.e.* the trees which are transformed) can be seen as an algebraic term. This idea, related to P. Wadler's views, allows Tom to rewrite any kind of data structure, and in particular XML trees, as long as a *formal anchor* is provided.

There are two possible approaches for using Tom. Either we start from an existing application that is improved with new functionalities implemented using the rewriting features of Tom, and in this case the data-structure used by the application is already defined: we just have to define a *mapping* from this data-structure to Tom.

Alternatively, we can abstract on the data-structure by using the data-structure generator integrated in Tom. Given a signature, Tom generates a set of Java classes that provide static typing. A subtle hash-consing technique is used to offer maximal sharing [8]: there cannot be two identical terms in memory. Therefore, the equality tests are performed in constant time, which is very efficient in particular when non-linear rewrite rules are considered.

In this section, we show how this general approach can be tailored to transform XML documents in both an expressive and a theoretically grounded way. Although the extension we present here is completely integrated in Tom we use the name TomML to refer to the syntax features and standard libraries specific to XML document manipulations.

For the rest of the paper we consider the following XML document:

```
<Bank name="BNP">
 <Branch name="Etoile">
  <CCAccount id="12">
   <Owner gender="M">Bob</Owner>
    <Balance>10000</Balance>
  </CCAccount>
   <SAccount rate="4">
    <Owner gender="M">Bob</Owner>
    <Owner gender="F">Alice</Owner>
    <Balance>100000</Balance>
   </SAccount>
 </Branch>
 <Branch name="Lafayette">
  <CCAccount id="23">
   <Owner gender="M">John</Owner>
   <Balance>10000</Balance>
  </CCAccount>
  <CCAccount id="6">
   <Owner gender="M">Bob</Owner>
   <Balance>6000</Balance>
  </CCAccount>
 </Branch>
</Bank>
```

A *bank* consists of several *branches*, each of them containing different types of *accounts*. Whatever the representation the XML document is (DOM for instance), it can be *seen* as a tree built out of (XML) nodes. For the scope of this paper we consider the following meta-model:

```
TNode = ElementNode(Name:String, AttrList:TNodeList, ChildList:TNodeList)
      | AttributeNode(Name:String, Specified:String, Value:String)
      | TextNode(Data:String)
      | CommentNode(Data:String)
      | CDATASectionNode(Data:String)
      | ...
TNodeList = concTNode(TNode*) // denotes a list of TNode
```

An `ElementNode` has a name and two children: a list of attributes, and a list of nodes. A *formal anchor* is materialized by a file which describes the Tom view of the corresponding XML document implementation[1] and, for the purpose of this paper, we have considered a correspondence between the algebraic sort `TNode` and the `Node` class from the `w3c.dom` package. This mapping is available in the standard TomML libraries and specifies, for example, how the name of an `ElementNode` of the Tom model can be retrieved (using the method `getNodeName` of the DOM class for instance). Once we have defined this mapping, the abstract notation can be used to match an XML document and print the name of all the branches of a bank:

```
%match(xmlDocument) {
 ElementNode("Bank",_,   //_ means that any list of attributes is accepted
  concTNode(_*,
   ElementNode("Branch",concTNode(_*,AttributeNode("name",_,bname),_*),_),
  _*)) -> { System.out.println("branch name: " + bname); }
}
```

We consider in this section that `xmlDocument` corresponds to the Tom encoding of the XML document given above and, in this case, the application of this code prints the strings “branch name: Etoile” and “branch name: Lafayette” and corresponds intuitively to the following XSLT template:

```
<xsl:template match="Bank/Branch">
    branch name: <xsl:value-of select="@name"/>
</xsl:template>
```

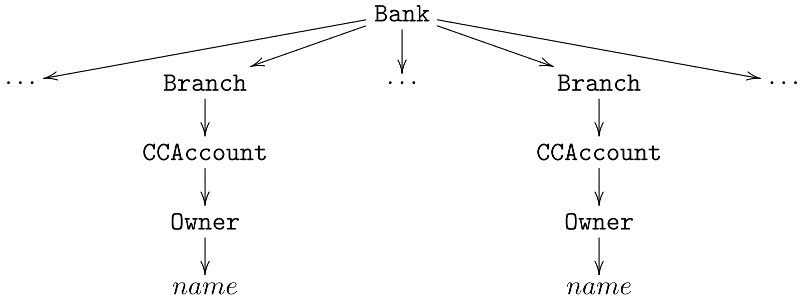
The interest of this approach is that the semantics of the match construct is theoretically well grounded and based on associative-matching with neutral element. The above pattern is rather complex partly because of the highly decorated XML syntax but Tom provides an alternative and much simpler XML tailored syntax. For example, the above match construct can be written:

```
%match(xmlDocument) {
 <Bank><Branch name=bname></Branch></Bank> -> {
  System.out.println("branch name: " + bname);
 }
}
```

Note that XML nodes can be directly used and that the extension variables (identified by “_*” previously) not used in the right-hand side are left implicit in the left-hand side. The semantics of these match constructs is exactly the one provided by Tom, and thus can deal with nested patterns, non-linear variables,

[1] see `http://tom.loria.fr` for more details

anti-patterns, *etc.*. For example, in order to print the *name* of all clients who own two credit-card accounts in two different branches of the bank, *i.e.* to match a template of the form:



a simple non-linear pattern can be used in TomML:

```
void ownerInTwoBranches(TNode xmlDocument) {
 %match(xmlDocument) {
  <Bank>
   <Branch name=bname1><CCAccount><Owner>name</Owner></CCAccount></Branch>
   <Branch name=bname2><CCAccount><Owner>name</Owner></CCAccount></Branch>
  </Bank> -> {
   System.out.println(name + " in " + bname1 + " and " + bname2);
  }
 }
}
```

As expected, this method prints the string “John in Etoile and Lafayette”. A similar behavior is obtained for the following XSLT code

```
<xsl:template match="Bank/Branch/CCAccount">
    <xsl:for-each select="preceding::CCAccount[Owner=current()/Owner]">
       <xsl:value-of select="Owner" /> in
       <xsl:value-of select="../@name" /> and
       <xsl:value-of
         select="following::CCAccount[Owner=current()/Owner]/../@name"/>
    </xsl:for-each >
</xsl:template>
```

which is clearly less intuitive than the corresponding Tom code and becomes even more elaborated when negative conditions like the ones below are needed.

The anti-patterns are convenient if we want to specify concisely definitions of relatively complex patterns implying negative conditions. The branches whose clients are all mutually different can be printed using the following method:

```
void branchWithNoMultipleOwner(TNode xmlDocument) {
 %match(xmlDocument) {
  <Bank>
   branch@!<Branch> <_><Owner>o</Owner></_>
                    <_><Owner>o</Owner></_> </Branch>
  </Bank> -> { printXMLFromTNode(branch); }
 }
}
```

The variable `branch` can be seen as alias for the whole term matched by the pattern following the "`@`" operator; this is of course just syntactic sugar allowing for concise definitions of the consequent actions. The function `printXMLFromTNode` available in the standard TomML libraries prints a `TNode` using an XML syntax.

The above pattern corresponds to an "all-different" constraint but other constraints like "all-equal" can be easily expressed.

## 4 Strategies

When programming using functions, pattern matching constructs, and more generally the notion of *transformation rule*, it is common to introduce extra functions that *control* their application. In the case of rewriting, this control describes how and when the rules should be applied. This control can be defined in the right-hand side of the rules but this is usually a bad practice since it makes the rules more complex, specific to a given application, and thus not reusable.

Rewriting based languages provide more abstract ways to express the control of rule applications, either by using reflexivity as in Maude, or the notion of strategy for Elan, Stratego [9], or ASF+SDF. Strategies such as *bottom-up*, *top-down* or *leftmost-innermost* are higher-order features that describe how rewrite rules should be applied. This compares to some extent to the "`//`" operator of XPath which corresponds to a *depth-search*. Tom offers a flexible and expressive strategy language where high-level strategies are defined by combining low-level primitives. Among these latter primitives, we consider the *sequence* (denoted `;`), the *choice* (denoted `<+`), and two generic congruence operators called `All` and `One`. A rewrite rule is also an elementary strategy that can be applied on a term (*i.e.* a tree).

As for the `%match` construct, a user defined strategy is defined by a pattern and an action but, additionally, it also specifies a default behaviour for the case when the pattern does not match. This default behaviour can be either the `Identity` meaning that no action is performed or `Fail` in which case an exception is raised when the pattern does not match. For example, when applying the strategy

```
%strategy printOwner() extends Identity() {
  visit TNode {
    <Owner gender="M">#TEXT(name)</Owner> -> {
      System.out.println(name);
    }
  }
}
```

to the term `<Owner gender="M">Bob</Owner>` the string "`Bob`" is printed, while applying it to `<CCAcount><Owner gender="M"> Bob </Owner></CCAccount>` leads to no action since the pattern does not match at the root position and the default behavior is the `Identity`.

The strategy can be fired on the variable `xmlDocument` by the Java statement `printOwner().visit(xmlDocument)`. It is important to understand that with such a statement the strategy is only applied on the root node of the corresponding document and that there is no automatic recursive application that would search for a convenient sub-tree. In Tom, the control is explicit and should be specified by an appropriate strategy built using the available strategy primitives. Nevertheless, such higher-level strategies can be easily defined and all the classical strategies are already available in the Tom standard library. For instance, the *top-down* strategy can be recursively defined by `TopDown(s)` $\triangleq$ `s;All(TopDown(s))` where `s1;s2` means that, first, `s1` is applied, and then `s2` is applied on the result of `s1`. It fails if `s1` or `s2` fails. The `All(s)` combinator applies `s` to all the immediate children of a given node. `TopDown(s)` corresponds thus to the application of `s` followed by a recursive application of `TopDown(s)` to *all* the immediate children. This strategy fails if the application of `s` fails. Note that the application of `All(s)` to a constant (*i.e.* a leaf of the tree) does not fail but it simply does nothing.

The execution of `TopDown(printOwner()).visit(xmlDocument)` prints the names of all the account owners in a bank (independently of the branch). Using other combinators such as `<+`[2] and `One`[3], it is easy to define other general purpose strategies such as `BottomUp`, `Innermost`, *etc.* More complex tasks can be accomplished by strategies using elaborated non-linear patterns involving (explicit) list matching. For example, if we want to update the initial document and give a 15% bonus to all account owners that have opened a savings account in the same branch then, the following strategy can be used:

```
%strategy bonus() extends Identity() {
    visit TNode {
       <Branch> (A1*,
                 <CCAccount>
                   (X1*,owner,X2*,<Balance>#TEXT(bal)</Balance>,X3*)
                 </CCAccount>,
                 A2*,
                 sa@<SAccount>owner</SAccount>,
                 A3*)
       </Branch> -> {
         TNode newbal =
           xml(<Balance>#TEXT(Double.parseDouble(bal)*1.15)</Balance>;
         return xml(<Branch> A1*
                     <CCAccount>X1* owner X2* newbal X3*</CCAccount>
                     A2* sa A3* </Branch>);
}   }  }
```

In this example we have used explicit lists of the form `(X1*,...,X2*,...,X3*)` to retrieve the context information (*i.e.* the other XML nodes) needed in order to build the XML tree in the right-hand side of the rule. The Tom construct `xml(...)` can be used to build a tree (a DOM object in our case) using an XML notation. Once again, the notation `sa@...` indicates that the matched node is stored in the variable `sa` and thus, this variable can be used in the right-hand side of the strategy. Due to lack of space, this has not be exemplified, but note

[2] `s1<+s2` tries to apply `s1`; if it succeeds, the result is returned, otherwise `s2` is applied

[3] `One(s)` searches for an immediate children where `s` can be applied

that the right-hand side of a rule is an arbitrary list of Java and Tom statements and therefore, recursive function calls as well as nested calls to strategies can be freely used.

## 5 Comparison with similar tools

There exist numerous languages aiming at manipulating XML documents. We briefly present some of them and emphasize the main differences with respect to the Tom approach.

XPath is a language providing a concise and efficient syntax for selecting parts of an XML document and querying XML documents. It can be used to describe the search of a particular node in the document in breadth and arbitrary depth. All XPath queries can be encoded within a Tom strategy. However, it is not possible with XPath to have full control over the way the document is explored, as it is with Tom strategies.

XSLT [10, 11] is a transformation language for XML, aiming at describing transformations from one XML dialect to another. It uses XPath to select part of the original document and query it, and offers only functional features, thus one can only loop over the results of an XPath query. The result of the application of an XSLT template on a document may only be another document. Contrary to Tom, it is not possible to execute arbitrary actions when examining the initial documents.

The OCamlDuce system [12] is a modified version of the OCaml functional language which integrates XDuce features, such as XML expressions, regular expression types and patterns, iterators. OCamlDuce fully integrates XML manipulations in the OCaml language, providing static type inference for XML expressions, by the mean of regular expression types. This provides a static insurance that a program will produce values of a given XML type. The integration of XML manipulation in Tom cannot provide such guarantee. On the other side, a main advantage of Tom is to be fully integrated in a Java environment.

The Java standard library provides a DOM implementation, that enables manipulation of XML documents through the DOM API. Additionally, the package `javax.xml.xpath` does provide an XPath implementation, that enables the Java programmer to evaluate XPath expressions over DOM documents. However, this approach is purely interpreted, and does not provide any guarantee on the transformation.

## 6 Conclusion

We have presented Tom, an extension of Java which adds support for algebraic data-types, pattern matching and strategic rewriting, focusing essentially on the XML related features of the language. The powerful pattern-matching construct of Tom allows one to express relatively complex matching conditions using concise and natural patterns. The strategies add more expressive power by providing a simple method for the traversing structured data. We should point out that all the rules as well as the corresponding guiding strategies should be explicitly given and thus no ambiguity concerning their application is possible.

Besides its strong expressive power and its solid semantics, the approach guarantees the portability of the applications that can be executed on top of any Java environment. We have mainly shown examples for querying XML documents but, as we have seen in the last example of Section 4, the `xml(...)` construct can be used to modify and build XML documents.

The applications developed in Tom are independent of the data structure implementation given that a mapping between the respective internal implementation and the Tom representation is given. The TomML standard libraries already provide this kind of mapping for DOM classes but new mappings can be easily integrated.

## References

1. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Eve Maler, F.Y., Cowan, J.: Extensible markup language (XML) 1.1 (second edition). Technical report, W3C (2006) http://www.w3.org/TR/2006/REC-xml11-20060816/.
2. Brand, M., Deursen, A., Heering, J., Jong, H., Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Wilhelm, R., ed.: Compiler Construction. Volume 2027 of LNCS., Springer-Verlag (2001) 365–370
3. Kirchner, H., Moreau, P.E.: Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. Journal of Functional Programming **11**(2) (2001) 207–251
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In Nieuwenhuis, R., ed.: Proceedings of RTA 2003. Volume 2706 of LNCS., Springer-Verlag (2003) 76–87
5. Moreau, P.E., Ringeissen, C., Vittek, M.: A Pattern Matching Compiler for Multiple Target Languages. In Hedin, G., ed.: 12th Conference on Compiler Construction, Warsaw (Poland). Volume 2622 of LNCS., Springer-Verlag (2003) 61–76
6. Balland, E., Kirchner, C., Moreau, P.E.: Formal islands. In: Proceedings of AMAST 2006. LNCS (2006) 51–65
7. Kirchner, C., Kopetz, R., Moreau, P.E.: Anti-pattern matching. In: Proceedings of the 16th European Symposium on Programming. Volume 4421 of LNCS., Springer Verlag (2007) 110–124
8. van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.: Efficient annotated terms. Software-Practice and Experience **30** (2000) 259–291
9. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, ACM Press (1998) 13–26
10. Kay, M.: XSL transformations (XSLT) version 2.0. Technical report, W3C (2007) http://www.w3.org/TR/2006/REC-xml11-20060816/.
11. Kay, M.: XSLT 2.0 and XPath 2.0 Programmer's Reference (Programmer to Programmer). Wrox Press Ltd., Birmingham, UK, UK (2008)
12. Frisch, A.: Ocaml + xduce. In Castagna, G., Raghavachari, M., eds.: PLAN-X, BRICS, Department of Computer Science, University of Aarhus (2006) 36–48