ATERMs for manipulation and exchange of structured data: it's all about sharing

Mark G.J. van den Brand♣ Paul Klint♠

♣ *Mathematics and Computer Science, Technical University of Eindhoven*
`www.win.tue.nl/~mvdbrand`

♠ *Centrum voor Wiskunde en Informatica (CWI), Software Engineering Department*
and
Informatics Institute, University of Amsterdam
`www.cwi.nl/~paulk`

26th September 2006

Abstract

Some data types are so simple that they tend to be reimplemented over and over again. This is certainly true for *terms*, tree-like data structures that can represent prefix formulae, syntax trees, intermediate code, and more. We first describe the motivation to introduce *Annotated Terms (*ATERMs*)*: unifying several term formats, optimizing storage requirements by introducing maximal subterm sharing, and providing a language-neutral exchange format. Next, we present a brief overview of the ATERM technology itself and of its wide range of applications. A discussion of competing technologies and the future of ATERMs concludes the paper.

1 History and Motivation

Some data types are so simple that they tend to be reimplemented over and over again. This is not only true for linked lists and symbol tables but also for *terms*, tree-like data structures that can represent prefix formulae, syntax trees, intermediate code, and more. The explanation is probably that every project needs slight variations of these simple data types and that existing parameterization techniques for software components cannot easily accommodate this variability.

Generic language technology is one of our research topics and related to this research we have developed an interactive development environment for writing language specifications, the ASF+SDF Meta-Environment [35, 8]. Terms play an important role in this Meta-Environment: they are used to represent source code, parse tables, error messages and so forth. When we made an inventory of term data types in our own software projects related to the ASF+SDF Meta-Environment, it turned out that we were using (and maintaining!) six different variants of a term data type and this provided a strong incentive to look for a single data type that could be used in all projects.

A first attempt at unification were the TOOLBUS terms that were introduced as part of the implementation of the TOOLBUS coordination architecture [3], our component interconnection technology. TOOLBUS terms introduced the simple *make-and-match* paradigm (explained below) for constructing and decomposing terms. A linear string representation was used to exchange terms between components. The C implementation supports automatic garbage collection.

Annotated Terms (or ATERMs as described in [11]) introduced several innovations over the original design: maximal subterm sharing, annotations, a compressed binary exchange format, and a two-level Application Programming Interface (API) that enables both simple and efficient use of ATERMs. Mature implementations exist for C and Java and experimental implementations for, C#, ML and Haskell.

Although ATERMs were introduced to solve just our own local problem, the wide acceptance of ATERMs in numerous projects suggests that this problem was not so local after all. The purpose of the present paper is to sketch the contexts and problem domains in which ATERMs are useful and to compare them with competing technologies. The plan of this paper is as follows. In Section 2 we give a quick introduction to ATERMs and discuss all technology that has been developed to seamlessly integrate ATERMs in applications. Next, we give a survey of applications of ATERMs in Section 3. We complete the paper with a comparison of ATERMs with other technologies (Section 4) and we speculate about their future (Section 5).

2 The ATERM Technology

2.1 A quick introduction to ATERMs

The data type of ATERMs is defined as follows (see [11] for full details):

- An integer or real constant is an ATERM.
- A function application is an ATERM, e.g., `f(a,b)`.
- A list of zero or more ATERMs is an ATERM, e.g., `[f(a),1,"abc"]`.
- A placeholder term containing an ATERM that represents the type of the placeholder is an ATERM, e.g., `f(<int>)`.
- A Binary Large OBject (BLOB) containing arbitrary binary data is an ATERM.
- A list of (*label*,*annotation*) pairs may be associated with each ATERM. Label and annotation are both ATERMs and can thus contain nested annotations.

ATERMs are constructed under the constraint that all subterms of all ATERMs in a given universe are *maximally shared*. ATERMs thus represent directed acyclic graphs and should, in fact, have been called "ADags". As a consequence, all operations on ATERMs are applicative: an ATERM can be decomposed into its constituent parts, but those parts can never be replaced. Replacement can only be achieved by building a new ATERM that contains new values at the places to be modified.

The ATERM API is based on the *make-and-match* paradigm:

- *make* (compose) a new ATERM by providing a pattern for it and filling in the placeholders in the pattern with given values.
- *match* (decompose) an existing ATERM by comparing it with a pattern and decompose it according to this pattern.

Functions for the input and output of ATERMs (both in textual and in binary form) are provided. For efficiency reasons also direct access functions for the constituents of ATERMs such as arguments of applications, elements of lists, and annotations are provided. As a first example, consider the following code fragment which shows how to *make* an ATERM (using the C version):

```
ATerm t1 = ATmake("or(true,false)");
ATerm t2 = ATmake("and(true,<term>)", t1);
```

First, the term `or(true,false)` is constructed and then assigned to variable `t1`. Next, a second term is constructed using the term pattern `"and(true,<term>)"`. The value of `t1` is substituted for the placeholder `<term>` and as a result the term `and(true, or(true,false))` is assigned to `t2`. Now let's try to *match* against this last term:

```
ATerm t3, t4;
if(ATmatch(t2, "and(<term>,<term>)", &t3, &t4)){
  ...
}
```

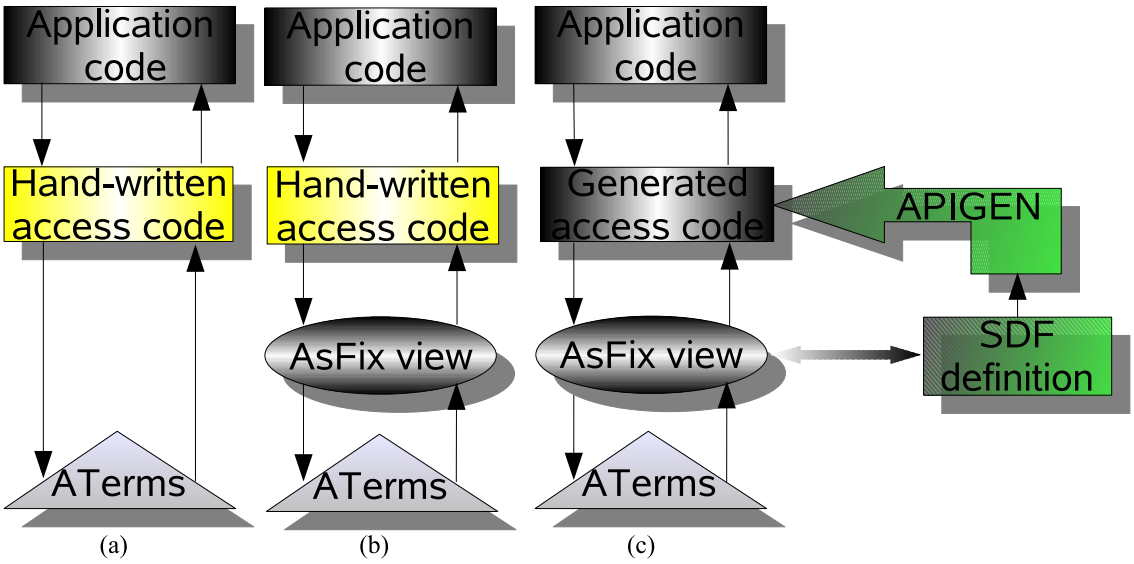

Figure 1: Application code uses ATERMs. (a) Application code uses hand-written code to manipulate ATERMs directly; (b) Application uses hand-written code to manipulate an ASFIX view on ATERMs; (c) The ASFIX view is defined in an SDF grammar and the access code is generated by APIGEN, the application uses this generated code to manipulate the ASFIX view.

The pattern "`and(<term>,<term>)`" is matched against the current value of `t2`. The match succeeds and the subterms corresponding to the placeholders, respectively `true` and `or(true,false)` are assigned to the variables `t3` and `t4`. The same example can also be coded using direct access to the term representation. For instance,

```
t3 = ATgetArgument(t2, 0);
t4 = ATgetArgument(t2, 1);
```

achieves the same effect as the `ATmatch` condition.

As these examples show, the physical structure of the terms being manipulated by this code is explicit in the form of patterns and indices representing argument positions. This intimacy between an application and the ATERM representation it uses is shown in Figure 1(a). The code would be broken by any change of the representation such as renaming function names (e.g., in a Dutch language version `and` might be replaced by `en`), swapping arguments, adding arguments, and the like. We will come back to the problem of representation hiding in Section 2.4.

2.2 Implementation of ATERMs

A comprehensive description of the ATERM implementation, including design decisions, data representation, algorithms and benchmarks is given in [11]. Here, we only want to highlight some specific aspects of the implementation.

2.2.1 Maximal subterm sharing

Our strategy to minimize memory usage is simple but effective: we only create terms that are *new*, i.e., that do not exist already. If a term to be constructed already exists, that term is reused, ensuring maximal sharing.The library functions that construct terms make sure that shared terms are returned whenever possible. The sharing of terms is thus invisible to the library user.

Maximal sharing of terms can only be maintained when we check at every term creation whether a particular term already exists or not. This check implies a search through all existing terms but must be

fast in order not to impose an unacceptable penalty on term creation. Using a hash function that depends on the internal code of the function symbol and the addresses of its arguments, we can quickly search for a function application before creating it. All terms are stored in a hash table. The hash table does not contain the terms themselves, but pointers to the terms. This provides a flexible mechanism of resizing the table and ensures that all entries in the table are of equal size. Hence the (modest but not negligible) cost at term creation time is one hash table lookup.

2.2.2 Garbage collection

The C implementation of the ATERM library offers, in addition to maximal subterm sharing, automatic garbage collection. This frees the software developer from all the effort of explicitly allocating and de-allocating terms. The garbage collector was initially based on the traditional mark-and-sweep garbage collector as developed by Boehm and Weiser [5]. However, one of the consequences of the maximal subterm sharing is that destructive updates of terms are not allowed. This functional behavior leads to the following *invariant*: the children of an ATERM are always older than the ATERM itself. Moreau and Zendra [45] have introduced a generational garbage collector in the ATERM library that exploits this invariant. The generational garbage collector is based on the assumption that newly created objects have on average a shorter life time than older objects. Older objects are thus inspected less frequently by the garbage collector. The observed efficiency gain for applications using the ATERM library is between 19% and 35%.

2.2.3 Java version

The Java implementation of the ATERM library provides a factory for creating ATERMs. It was designed to support two implementations: one that directly uses the C version of ATERMs and one written in pure Java. Only the latter has been actually implemented. It uses SharedObjects, a generic mechanism for maximal subterm sharing in Java [17].

2.3 Application-specific ATERM views: ASFIX and family

ATERMs provide a simple, untyped, representation for structured data. In the application domains where ATERMs serve as representation for syntax trees, the need arises, however, to encode information regarding syntactic categories in the tree representation itself. Typically, all nodes in the syntax tree are represented by a binary application operator `appl`. Its first argument is the tree representation of the syntax rule used to parse the construct and the second argument is the list of children of the construct (possibly including concrete textual information, see below). This syntax tree representation is thus self-describing but seems also very redundant since descriptions of syntax rules are repeated for every use of that rule. However, thanks to maximal subterm sharing, this causes no problems. It should be stressed that this representation can still be viewed as an ordinary ATERM.

Another design choice is whether to represent *parse trees* or *abstract syntax trees*. The former represent the source text including all textual aspects such as comments and layout while the latter focus on hierarchical structure but disregard textual information. Parse trees are better suited for applications in the areas of source code analysis and renovation, while abstract syntax trees suffice for compilation-oriented applications. In the following we focus on parse trees.

The first dedicated format we designed along these lines was ASFIX [36, 63, 7], a fixed format originally intended for representing the parse trees of ASF specifications but in fact capable to represent arbitrary parse trees. ASFIX defines a specific view of the application code on the underlying ATERMs as is shown in Figure 1(b). Extensive examples of this format are given in [32]. By now several other systems use their own variants of ASFIX.

Continuing our example from Section 2.1, we explore how to represent the parse tree corresponding to the Boolean expression `true & false`. First, we have to assume the existence of a grammar for a fragment of the Boolean expressions:

```
Bool ::= "true"
Bool ::= "false"
```

```
Bool ::= Bool "&" Bool
...
```

Since we do not want to go into the details of representing grammar rules, we just assume that $<< R >>$ yields the term representation of the corresponding grammar rule *R*. The parse tree for `true` will now look like

```
appl(<<Bool ::= "true">>, ["true"])
```

while the parse tree for `true & false` will look like:

```
appl(<<Bool ::= Bool "&" Bool>>,
     [ appl(<<Bool ::= "true">>, ["true"]),
       " ",
       "&",
       " ",
       appl(<<Bool ::= "false">>, ["false"])
     ])
```

Observe how the second argument of the application operator is a list of subtrees separated by the layout between these subtrees. The following observations can be made about this representation:

- The above representation is an ordinary ATERM.
- This ATERM is also a valid ASFIX term since in each application node the list of arguments is compatible with the grammar rule in the first argument.
- Accessing the arguments of this ASFIX term can be done using ordinary ATERM operators.
- This access becomes involved and prone to changes in the grammar.

2.4 Generating access code for ATERMs and ATERM views using APIGEN

2.4.1 ASFIX view

We now have two stacked representation layers for our syntax trees. The bottom layer are pure, untyped, ATERMs and the top layer is the ASFIX view just described: ATERMs that provide a self-describing view on the underlying ATERM representation.

Accessing ASFIX terms can be done in two ways: using high-level match primitives or using low-level access functions which are both provided by the ATERM library. This has three major disadvantages:

- For complex terms, the access code becomes hard to read or modify.
- Since the underlying ATERM representation is untyped, there is no guarantee that the access code is type safe.
- If the underlying grammar of ASFIX (or one of its variants) changes, the access code becomes incorrect.

We have solved this problem by *generating* the access code instead of writing it manually, using a generator called APIGEN. The key idea as described in [32] is to use a syntax definition (written as a grammar in SDF [28, 62, 63] or as an Abstract Data Type, explained below) of the desired syntax tree format and to generate fully typed access code (in C) from this syntax definition. The SDF definition has to be decorated with constructor information and labels in order to distinguish between alternative production rules and the members within a production rule, respectively. In this way, half of the hand-written access code can be replaced by automatically generated code, thus enhancing type-safety, efficiency and modifiability. Figure 1(c) illustrates the role that APIGEN plays in the relation between application code and ATERMs.

In [17] the APIGEN approach is refined and extended to Java as well. A special challenge here is how to achieve maximal subterm sharing and effective garbage collection in Java.

```
[
  [Graph, default, graph(<nodes(NodeList)>,<edges(EdgeList)>,
                         <attributes(AttributeList)>)],
  ...
  [NodeList, empty, []],
  [NodeList, multi, [<head(Node)>,<[tail(NodeList)]>]],
  ...
  [Style, bold, bold],
  ...
  [NodeId, default, <id(str)>],
  ...
  [Color, rgb, rgb(<red(int)>,<green(int)>,<blue(int)>)],
  ...
]
```

Figure 2: Part of an ADT to represent graphs.

```
Graph makeGraphDefault(NodeList nodes, EdgeList edges,
                       AttributeList attributes);
NodeList makeNodeListEmpty();
NodeList makeNodeListMulti(Node head, NodeList tail);
NodeId makeNodeIdDefault(char* id);
Color makeColorRgb(int red, int green, int blue);
Style makeStyleBold();
```

Figure 3: A subset of the generated constructor functions.

2.4.2 Abstract data type view

In addition to SDF as structure definition, APIGEN also accepts an abstract data type description (ADT). For simplicity, the description format of this ADT is an ATERM as well. Figure 2 shows part of an ADT for representing graphs and illustrates the most important elements that can be used when writing an ADT. An ADT entry consists of a *type* (`Graph`, `NodeList`, `Style`, `NodeId`, `Color`), a *constructor* (`default`, `empty`, `multi`, `bold`, `rgb`), and a *pattern*. The constructor name can be any name as long as all alternatives for a type have a unique constructor name. If there is only one pattern for a type, we use the constructor name `default`. The pattern is an ATERM that may contain placeholders that serve as arguments of the pattern. The general format of an argument is `<label(type)>`. This indicates that the argument matches a subterm of type `type`. The *`label`* should be unique within a pattern and provides an access mechanism to the subterm.

The first entry of Figure 2 (`Graph`) represents a pattern with three arguments. The second and third entry (`NodeList`) describe the list pattern. The fourth entry (`Style`) represents a pattern without arguments and the last two patterns (`NodeId` and `Color`) represent the basic types `str` and `int`, respectively. Given such ADT a collection of functions, constructors, getters, setters, etc., is generated which allow a type-safe manipulation of the underlying ATERMs. See Figure 3 for some of the generated constructor functions based on the ADT of Figure 2.

The generated API allows the manipulation of abstract-syntax-tree-like terms, again in a type-safe way. The ATERMs access APIs based on these abstract data type descriptions are intensively used in the Meta-Environment, not only to manipulate ASFIX terms, but also to construct and manipulate error messages, parse tables, and the like.

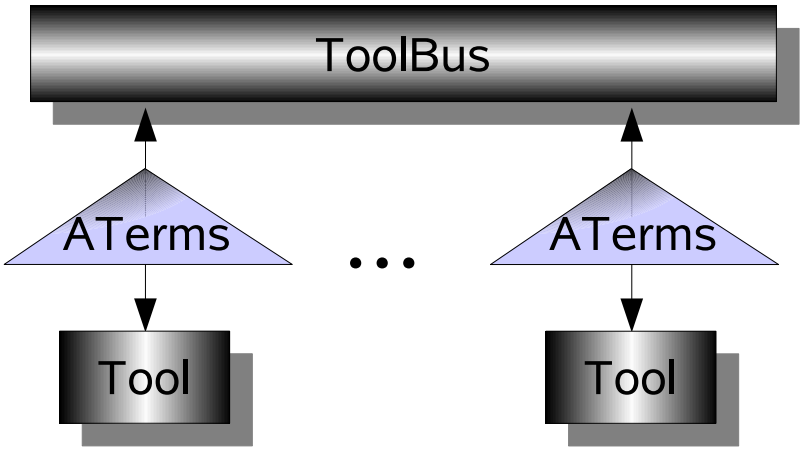

Figure 4: The TOOLBUS coordination architecture.

3 Applications of ATERMs

3.1 Component-based development and interoperability

As was already mentioned in the introduction, ATERMs were originally introduced as part of the TOOLBUS [3] coordination architecture as shown in Figure 4. Key idea is that software components ("tools" in TOOLBUS parlance) written in different implementation languages are connected via the TOOLBUS that acts as language neutral middleware and coordinator. Unlike older technologies like DCOM and CORBA, the TOOLBUS uses a centralized description of the cooperation between the components that form the application and can thus enforce the workflow in the application and control the interaction between components. The communication between TOOLBUS and tools takes place in the form of ATERMs and thus enables interoperability between languages and run-time or hardware platforms. The software engineering aspects of this approach are further elaborated in [37, 24]. Note that maximal subterm sharing in ATERMs is restricted to a single component/machine environment: in other words, a subterm of an ATERM can only point to ATERMs in the same environment and not to ATERMs in another component/machine environment. However, when an ATERM is serialized and shipped to another environment, its internal maximal subterm sharing is preserved.

One of the key applications of the TOOLBUS is the ASF+SDF Meta-Environment, an Interactive Development Environment for writing and generating language-related tools. These tools may perform tasks ranging from supporting a domain-specific language to performing analysis and transformation of a large software system. The ASF+SDF Meta-Environment itself contains many components such as a user-interface, parser and unparser generator, compiler, interpreter, and more that are interconnected via the TOOLBUS. ATERMs are used for a variety of tasks:

- Abstract data type definitions, the intermediate data format used by APIGEN as described earlier (see Section 2.4).
- Configuration files.
- Parse tables as produced by a parser generator and used by the parser.
- Syntax trees produced by the parser. This regards syntax trees of ASF+SDF specifications as well as syntax trees of source texts in arbitrary languages such as C, Java or Cobol.

Based on this technology, similar development environments have been built for ELAN [6], Action Semantics [10], and CHI [2]. The extension mechanisms needed to achieve this are described in [16].

3.2 Implementation of term rewriting languages and engines

ATERMs are being used in the implementation of various languages such as Action Semantics [56], ASF [9, 14], CASL [12, 18, 52, 48], ELAN [6], FERUS [42, 46, 47], JITty [50], μCRL [4, 41], Strafunsky [40], and Stratego [19].

ATERMs are also used in related language implementation tools such as JJForester [39], a Java implementation of the visitor pattern, TOM [44], a generator for embedded pattern matching code in Java, and others. GOM [51] combines TOM functionality with a description of abstract syntax trees. This combination is used to generate Java implementations for these trees. This provides a concise way to maintain the internal representation in a canonical form with respect to the specified equational theory, in the form of rewrite rules. These rewrite rules automatically keep the corresponding nodes in normal form.

The use of annotations is discussed in [38], while [15] describes the role of ATERMs for disambiguation during parsing.

Typically, ATERMs are used as run-time representation for terms or trees that are needed during the execution of these languages and tools. The main advantages are:

- Maximal subterm sharing reduces the memory foot print during execution. In [9] this effect is measured in the context of term rewriting.
- An additional benefit of maximal subterm sharing is that a deep (structural) equality test between terms can be replaced by one pointer comparison. This reduces execution time.
- The annotations that can be attached to ATERMs have many applications in language implementations and range from source code coordinates to symbol table information.
- ATERMs can be exchanged with tools written in different implementation languages and thus promote reuse of existing tools.

In [13] a general perspective is sketched of the applications of term rewriting (and thus of ATERMs) in software engineering.

3.3 Source code representations and software renovation

In the reverse engineering community many research groups are involved in building tools for reverse engineering tasks such as parsing, analyzing, and transforming source code or visualizing results. The effort to build these tools is considerable, so it is preferable to re-use tools built by other groups. This can only be achieved if these tools have a common format to exchange source code or derivates of source code, e.g., control flow graphs or data flow graphs. ATERMs have been considered as an option to be used as exchange format [54], however many reverse engineering projects have decided to base their exchange format on XML [29, 54, 30, 33, 34, 31, 67, 43, 21]. There were several reasons for doing this. First of all, there was a strong need to represent graphs and one concluded that this can be done better in XML although graphs can be represented in ATERMs as well. Secondly, the overall feeling was that XML was the default standard for data exchange and by catching up on this technology more XML processing tools would become freely available. It is unclear whether, at the time of writing, this expectation has been fulfilled, although the community of developers of XML tools is much bigger than the community of ATERM developers.

One of the drawbacks of using an XML-like data format is the sheer size of the data that have to be exchanged between various tools. The fact that GXL is not suited for representing huge parse trees of industrial applications is not an issue in a research context, but this is crucial when performing large scale transformations on source code as we can learn from the following example.

At Bell Labs, a tool suite has been developed to perform transformations on C++ source code [66]. It uses, amongst others, ATERMs, SDF and Stratego [19] as technologies. One of the requirements in this project was that the original comments and layout should be preserved as much as possible when transforming the C++ source code. This implied the use of parse trees instead of abstract syntax trees. The C++ programs that had to be transformed where in the range of 100,000 upto 500,000 lines of code. An XML-based representation would have been infeasible and maximal subterm sharing is crucial to obtain

manageable terms. Clearly, the goal of this project was to come up with a tool suite to be used by Bell maintenance programmers and not to create an open exchange and transformation framework.

Another application of ATERMs in the area of program transformation is the restructuring of Cobol legacy code [58, 59, 60]. This work is focused on improving the maintainability of Cobol code which can only be achieved if the original layout and comments are preserved. The actual restructuring operations are expressed as rewrite rules in ASF+SDF. The ASF interpreter applies these rewrite rules on the parse trees of the Cobol sources. One of the strong characteristics of the ASF interpreter is rewriting with layout [61]. The transformations have been applied to large Cobol systems consisting of more than 1 MLOC spread over almost 1,000 separate Cobol source files. Once again, the need for the parse trees to contain all the original layout and comments in combination with maximal subterm sharing provided by the ATERM library made this project feasible.

3.4 Representing Web ontologies

OWL (Web Ontology Language) [49] is used to represent ontologies for the semantic web. Applications developed for the Semantic Web need some form of reasoning. Description Logics (DL) provide reasoning algorithms, however existing algorithms are less suited for being applied in Semantic Web applications. Pellet [55] is an OWL-DL reasoner written in Java specific for Semantic Web applications. The various components of Pellet are built on top of the ATERM library. The ATERM library is very suited to represent complex OWL class expressions since subterm sharing reduces the overall memory consumption spent for storing concept expressions and makes it is easy to transform the data from Pellet to external APIs.

3.5 Representing state spaces and feature diagrams

A Binary Decision Diagram (BDD) is a directed acyclic graph used to represent a Boolean function. Each node in the graph corresponds to a Boolean variable and has two outgoing edges: one for the true case and one for the false case. See [20] for a survey of BDDs and their applications in verification and model checking.

Since ATERMs also represent directed acyclic graphs, it comes as no surprise that they can be used to represent BDDs efficiently. This has been done for *model checking* [27] where the BDD represents a property of the state space of some system under investigation. Since real systems have a huge number (i.e., over 10^9) of states a concise representation of the state space and its efficient manipulation are mandatory. This typically leads to ATERMs in the Giga-byte range. The haRVey system [22] is a BDD-based satisfiability checker that also uses ATERMs as internal representation format.

Another application area are *feature diagrams* used for representing the variability of software systems. It turns out that even seemingly modest systems may have billions of variations. ATERMs have been used to check the consistence of feature diagrams [23, 57].

4 ATERMs compared with other technologies

How do ATERMs compare to other technologies, in particular XML? While making a comparison, we will discuss some unique characteristics of ATERMs.

4.1 XML

Although ATERMs and XML have many common properties, the motivations behind them do not overlap so well. Very roughly, the purpose of XML is to provide an easy and extensible exchange format, whereas ATERMs is a very efficient representation format for some specific data types. This makes them not truly comparable. For instance, in principle there is nothing that prevents XML from having maximal subterm sharing.

Since the question "why don't you guys just use XML like everybody else?" is on top of our list of frequently asked questions, we will answer it in some detail.

Sharing and compression XML provides a very weak form of subterm sharing. It is possible to use a special kind of reference tags of the form `ID=<name>` and `IDREF=<name>` in order to mark a subterm and to point to the marked subterm, respectively. Through this mechanism it is possible to mimic subterm sharing in XML. Upon construction of the XML term a possible candidate for sharing has to be recognized and a mark must be added to this term, via the tag `ID` and a unique name. Whenever, the same term is encountered later on, a reference is created via `IDREF` and the corresponding unique name. This puts quite a burden on the tooling, a table has to be maintained with subterms and unique reference identifiers, a mechanism must be available to recognize shared subterms, etc. This way of maintaining subterm sharing is very explicit and visible, whereas the maximal subterm sharing is invisible while using the ATERM library. Of course, the implementation of the ATERM library uses an internal table to administrate subterm sharing but it is hidden for the user.

When converting (serializing) an ATERM to an external exchange format the sharing is automatically preserved in the result by encoding. There are three different ways of serializing an ATERM:

- Binary encoded with maximal subterm sharing; this format is unreadable for humans. This way of serializing terms is only available in the C version of the ATERM library.
- Textually encoded with maximal subterm sharing; this format is human readable. Both the C version and Java version of the ATERM library support this way of serializing terms.
- Textually encoded without maximal subterm sharing; this format is again human readable but leads in many cases to a size explosion of the resulting text. Both the C version and Java version of the ATERM library support this way of serializing terms.

We used this latter facility to make a comparison between terms with maximal subterm sharing and `gzip`, an open source compression tool for Linux. In [11] we report on the obtained results on a given test suite. Using compression via maximal subterm sharing and binary encoding results in 85% compression whereas using non-shared terms and applying `gzip` results in 92% compression. The gain of using a dedicated binary representation of ATERMs is thus marginal.

Compression tools, like `gzip`, are frequently used to reduce the size of XML as well. It is unclear how the above experiments carry over to XML.

Note that the above discussion is *only* about the compression that can be achieved in the serialized version of an ATERM when it is sent to another component. The advantages of maximal subterm sharing *during* the processing of an ATERM inside a component are evident and unaffected by the above discussion.

DTD and XML schemas ATERMs are primarily intended for environments of closely cooperating components that need efficient data exchange. In such an environment it is acceptable that the cooperating components share some common knowledge about the format of the data. This differs from data exchange between loosely cooperating components as occurs in general web services and service-oriented architectures or, more specifically, in collaborative reverse engineering research projects. In such environments the data have to be fully self-contained.

The parse tree representation, ASFIX, as described in Section 2 is an example of a *semi self-contained* representation. An ASFIX tree contains all relevant information, such as production rules, lexical entities, layout, to either reconstruct the original text or to construct the parse tree. However, the meta information—what is a production rule, what is a tree node, and the like—is not encoded in the term itself. This information is described in a separate ADT, which is used to generate a library to manipulate these ASFIX trees. It is precisely this knowledge that has to be shared by the cooperating tools.

In the case of XML, a DTD or schema [65] can be used to describe that information as well and the resulting XML data are thus *fully self-contained*, i.e., components do not have to share any other knowledge in order to be able to cooperate. The need for this generality depends on the intended applications.

APIGEN versus DOM and SAX Given a structure definition in the form of either an SDF definition or an ADT, APIGEN generates an API to manipulate the ATERMs according to that structure. These ATERMs should reside entirely in memory. This closely resembles the tree-based APIs generated via the Document

Object Model (DOM). DOM [64] is a platform and language neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of an XML document. The difference between an API generated by APIGEN and an API generated by DOM is that APIGEN generates an API in C or Java to manipulate the underlying ATERMs directly and DOM is language independent and does not describe how the XML terms are internally represented.

An alternative for manipulating XML terms is SAX (Simple API for XML) [53] which uses an event-based API. The APIs generated by SAX allow the manipulation of XML terms without the need for building an internal tree. A call back mechanism is used while parsing the XML term. This reduces memory and CPU usage considerably. The APIs generated by APIGEN are tree-based and not event-based as in the SAX approach. Since ATERMs are much smaller than the internal representation of XML data, the need for an event-based API has not (yet) arisen.

4.2 Other intermediate formats

Giving a comprehensive overview of intermediate formats is beyond the scope of this paper, but we want to further stress the point that there is room for intermediate formats other than XML.

YAML An example of another popular intermediate format is YAML (YAML Ain't Markup Language), a "straightforward machine parsable data serialization format designed for human readability and interaction with scripting languages such as Perl and Python. YAML is optimized for data serialization, configuration settings, log files, Internet messaging and filtering. YAML(tm) is a balance of the following design goals:

- YAML documents are very readable by humans.
- YAML interacts well with scripting languages.
- YAML uses host languages' native data structures.
- YAML has a consistent information model.
- YAML enables stream-based processing.
- YAML is expressive and extensible.
- YAML is easy to implement." [68]

YAML and ATERMs are largely identical in goals and philosophy except that YAML takes the position of text markup and human readability while ATERMs use prefix terms with comma-separated argument lists and favor machine readability and sharing. It would be trivial to make a mapping between the two formats, except that the maximal subterm sharing of ATERMs cannot be preserved in a mapping to YAML.

Related approaches In [1], the Factotum system is described: "Factotum is a software system for implementing symbolic computing systems on DAG-based structures that critically rely on sharing of equivalent subterms. It provides a subterm sharing facility that is automatic and systematic, analogously to the way that automatic memory management is provided by a garbage collector." The system provides an API for creating and maintaining in-core representations of tree-like structures that may contain application-specific labels. The sharing of subterms is, however, not maximal but based on a Sharer subsystem that continuously tries to improve the amount of sharing in the tree. There is also no serialization facility to exchange data between different components.

5 The Future of ATERMs

ATERMs were created to reduce the number of different term representations in the ASF+SDF Meta-Environment project. The maximal subterm sharing and automatic garbage collection motivated other researchers to jump on the ATERM train and to contribute. The development of an API generator, which

allows type safe manipulation of ATERMs without loss of efficiency ensured an agile way of developing applications based on the ATERM library. There is a clear need for ATERMs, being a light-weight approach to the efficient exchange of structured data.

The XML community could profit from ATERMs if we would introduce mappings from and to XML. However, just introducing this mapping on the term level is not sufficient.

- Going from ATERMs to XML also involves the translation of the corresponding SDF definition or ADT definition to a DTD or XML schema. This is necessary in order to allow other XML tools to perform the validation.
- Going from XML to ATERMs involves the translation of the corresponding DTD or XML schema to a structure description which can be used by APIGEN in order to generate a library for manipulating the generated ATERM in a type-safe way. SAX allows an efficient way of constructing the corresponding ATERM, but it does not allow a type-safe manipulation of the constructed ATERM.

Although experimental mappings between ATERMs and XML have been implemented, this remains a mostly unexplored area.

Versions of the ATERM library exist for the two mainstream languages C and Java. It would help to have consolidated ATERM libraries for languages like Python, Ruby and C# that are rapidly becoming more popular.

Thanks to the applicative nature of ATERMs they are ideally suited for execution in a concurrent environment. A final, exciting, perspective is the development of massively parallel ATERM servers as enabled by the algorithms described in [26, 25]. Continuing the line of thought started in Sections 3.5 and 4, applications like concurrent model checking, concurrent feature analysis and concurrent ontology processing become within reach. This is also true for concurrent software analysis: running many concurrent analyses on the same syntax tree will lead to radically new forms of analysis.

In the end, it's all about sharing: the technical advantages of maximal subterm sharing combined with the shared implementation effort of ATERMs and related tooling enable applications in a variety of, unanticipated, application areas.

Credits and Acknowledgments

TOOLBUS terms were first implemented by Paul Klint in C. Pieter Olivier introduced the idea of maximal subterm sharing and started new C and Java implementations together with Hayco de Jong. The Java implementation was further improved by Pierre-Etienne Moreau by introducing SharedObjects (a generic mechanism for maximal subterm sharing in Java) and by improving garbage collection. APIGEN for C was developed by Hayco de Jong and Pieter Olivier. A new version of APIGEN for C and Java was developed by Mark van den Brand, Pierre-Etienne Moreau and Jurgen Vinju. We thank all users of ATERMs for their help, contributions and positive feedback. We thank the anonymous referees for their helpful feedback on this paper.

From `www.aterm.org` or `www.meta-environment.org` you can download ATERMs, APIGEN and several other systems mentioned in this paper.

References

[1] D.J. Sherman A1 and N. Magnier. Factotum: Automatic and systematic sharing support for systems analyzers. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98,*, volume 1384 of *Lectures Notes in Computer Science*, pages 249–262. Springer, March/April 1998.

[2] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*, 2006. To appear.

[3] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[4] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL a toolset for analysing algebraic specifications. In *Proceedings 13th Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lectures Notes in Computer Science*, pages 250–254. Springer, 2001.

[5] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18:807–820, 1988.

[6] P. Borovanský, C. Kirchner, H. Kirchner, P.E. Moreau, and C. Ringeissen. An overview of ELAN. *Electronic Notes in Theoretical Computer Science*, 15, 1998.

[7] M.G.J. van den Brand, H.A. de Jong, and P.A. Olivier. A common exchange format for reengineering tools based on aterms. In *Workshop on Standard Exchange Formats (WoSEF) at (ICSE'00)*, 2000.

[8] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[9] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[10] M.G.J. van den Brand, J. Iversen, and P.D. Mosses. An action environment. *Science of Computer Programming*, 61(3):245–264, 2006.

[11] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[12] M.G.J. van den Brand, P. Klint, and P.A. Olivier. ATerms: Exchanging data between heterogenous tools for CASL. Technical Report T-3, CoFi, March 1998.

[13] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications, WRLA 98*, 1998.

[14] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[15] M.G.J. van den Brand, A.S. Klusener, L. Moonen, and J.J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. In B. Bryant and J. Saraiva, editors, *Proceedings of the Third Workshop on Language Descriptions*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[16] M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. Environments for term rewriting engines for free! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lectures Notes in Computer Science*, pages 424–435, 2003.

[17] M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. A generator of efficient strongly typed abstract syntax trees in java. *IEE Proceedings-Software*, 152(2):70–78, 2005.

[18] M.G.J. van den Brand and J. Scheerder. Development of parsing tools for CASL using generic language technology. In P.D. Mosses D. Bert, Ch. Choppy, editor, *Recent Trends in Algebraic Development Techniques: 14th International Workshop, WADT 99*, volume 1827 of *Lectures Notes in Computer Science*, pages 89 – 105. Springer, 2000.

[19] M. Braveboer, K.T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for transformation systems. In *PEPM06*, pages 95–99, 2006.

[20] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[21] V. David. Attribute grammars for C++ disambiguation. Technical Report 0405, Laboratoire de Recherche et Développpement de l'Epita, 2004.

[22] D. Deharbe and S. Ranise. Light-weight theorem proving for debugging and verifying units of code. In *First International Conference on Software Engineering and Formal Methods (SEFM'03)*, pages 220–229. IEEE Computer Society Press, 2003.

[23] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.

[24] A. van Deursen and L. Moonen. From research to startup: Experiences in interoperability. In J. Ebert, K. Kontogiannis, and A. Winter, editors, *Interoperability in Reengineering Tools*, volume 296 of *Dagstuhl Seminar Report*, 2001.

[25] H. Gao, J.F. Groote, and W.H Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 17:21–42, 2005.

[26] H. Gao, J.F. Groote, and W.H Hesselink. Lock-free parallel garbage collection. In Y. Pan, D. Chen, M. Guo, J. Cao, and J. Dongarra, editors, *Proceedings of Third International Symposium on Parallel and Distributed Processing and Applications (ISPA'05)*, number 3758 in Lectures Notes in Computer Science, pages 263–274, 2005.

[27] J. F. Groote and J. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000*, volume 1955 of *Lectures Notes in Computer Science*, pages 161–178. Springer, 2000.

[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.

[29] R. C. Holt, A. Winter, and A. Schurr. GXL: Toward a standard exchange format. In K. Kontogiannis and F. Balmas, editors, *Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171. IEEE, 2000.

[30] D. Jin. Exchange of software representations among reverse engineering tools. Technical Report ISSN-0836-0227-2001-454, Department of Computing and Information Science, Queen's University, Kingston, Ontarion, Canada, 2001.

[31] D. Jin, J.R. Cordy, and T.R. Dean. Where's the schema? a taxonomy of patterns for software exchange. In *International Workshop on Program Comprehension (IWPC)*, pages 65–74. IEEE, 2002.

[32] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 50(4):35–61, 2004.

[33] H.M. Kienle. Exchange format bibliography. *Software Engineering Notes*, 26(1):56–60, 2001.

[34] H.M. Kienle, J. Czeranski, and Th. Eienbarth. The API perspective of exchange formats. In *Workshop on Standard Exchange Format (WoSEF)*, pages 33–39, 2000.

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

[36] P. Klint. Writing meta-level specifications in ASF+SDF. Centrum voor Wiskunde en Informatica, Unpublished, 1994.

[37] 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.

[38] J. Kort and R. Laemmel. Parse-tree annotations meet re-engineering concerns. In *Proc. Source Code Analysis and Manipulation (SCAM'03)*, pages 161–172. IEEE, 2003.

[39] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. *Science of Computer Programming*, 47(1):59–87, 2003.

[40] R. Lämmel and J. Visser. A Strafunski application letter. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *Lectures Notes in Computer Science*, pages 357 – 375. Springer, 2003.

[41] I.A. van Langevelde. A compact file format for labeled transition systems. Technical Report SEN-R0102, Centrum voor Wiskunde en Informatica, 2001.

[42] G. Lima, A.M. Moreira, D. Deharbe, D. Pereira, and D. Sena. FERUS: Um ambiente de desenvolvimento de especificacoes CASL. In *Proceedings of 16th Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 1–6, 2002. (In Portuguese)).

[43] E. Mamas and K. Kontogiannis. Towards portable source code representations using XML. In *Proceedings 7 th Working Conference on Reverse Engineering(WCRE-2000)*, pages 172–182. IEEE, 2000.

[44] P.E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *Compiler Construction: 12th International Conference, CC 2003*, volume 2622 of *Lectures Notes in Computer Science*, pages 61–76. Springer, 2003.

[45] P.E. Moreau and O. Zendra. GC^2: a generational conservative garbage collector for the atimage library. *Journal of Algebraic and Logic Programming*, 59(1–2):5–34, 2003.

[46] A.M. Moreira and A.S. de Oliveira. Simulating algebraic specification genericity on languages with initial semantics. In *WMF03*, Electronic Notes in Theoretical Computer Science. Elsevier, 2003.

[47] A.M. Moreira, C. Ringeissen, D. Déharbe, and G Lima. Manipulating algebraic specifications with term-based and graph-based representations. *Journal of Logic and Algebraic Programming*, 59(1–2):63–87, 2003.

[48] T. Mossakowski. CASL: From semantics to tools. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000*, volume 1785 of *Lectures Notes in Computer Science*, pages 93–108. Springer, 2000.

[49] *OWL*. Available at: `http://www.w3.org/TR/owl-guide/`.

[50] J. van de Pol. JITty: A rewriter with strategy annotations. In S. Tison, editor, *Rewriting Techniques and Applications : 13th International Conference, RTA 2002*, volume 2378 of *Lectures Notes in Computer Science*, pages 367–370. Springer, 2002.

[51] A. Reilles. Canonical abstract syntax trees. In *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*. Electronic Notes in Theoretical Computer Science, 2006. to appear.

[52] D. Sanella. The common framework initiative for algebraic specification and development of software: Recent progress. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques: 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting*, volume 2267 of *Lectures Notes in Computer Science*, pages 328–344. Springer, 2002.

[53] *Simple API for XML (SAX)*. Available at: `http://www.saxproject.org/`.

[54] S.E. Sim and R. Koschke. WoSEF: Workshop on standard exchange formats. *Software Engineering Notes*, 26(1):44–49, 2001.

[55] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. Submitted for publication to Journal of Web Semantics.

[56] T. van der Storm. Implementing actions. Master's thesis, University of Amsterdam, 2003.

[57] T. van der Storm. Variability and component composition. Technical Report SEN-R0403, Centrum voor Wiskunde en Informatica, 2004.

[58] N. Veerman. Revitalizing modifiability of legacy assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.

[59] N. Veerman. Towards lightweight checks for mass maintenance transformations. *Science of Computer Programming*, 57(2):129–163, 2005.

[60] N. Veerman and E. Verhoeven. Cobol minefield detection. *Software: Practice and Experience*, 2006. To appear.

[61] J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, University of Amsterdam, November 2005.

[62] E. Visser. A family of syntax definition formalisms. In M.G.J. van den Brand et al., editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[63] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[64] *W3C XML Document Object Model (XML DOM)*. Available at: `http://www.w3c.org/DOM`.

[65] *W3C XML Schema*. Available at: `http://www.w3c.org/XML/Schema`.

[66] D.G. Waddington and B. Yao. High-Fidelity C/C++ Code Transformations. In *Fifth Workshop on Language Descriptions, Tools, and Applications 2005 (LDTA 2005)*, pages 35–56. Elsevier, 2005.

[67] A. Winter. Exchanging graphs with GXL. In P. Mutzeland, M. Jünger, and S. Leipert, editors, *Graph Drawing : 9th International Symposium, GD 2001*, volume 2265 of *Lectures Notes in Computer Science*, pages 485–500. Springer, 2002.

[68] *YAML*. Available at: `http://www.yaml.org/`.