Generation of Abstract Programming Interfaces from Syntax Definitions

H.A. de Jong[1] and P.A. Olivier[2]

CWI, Department of Software Engineering,
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
[1]`Hayco.de.Jong@cwi.nl`
[2]`Pieter.Olivier@cwi.nl`

Abstract

This paper describes how an Abstract Programming Interface (API) and its implementation can be generated from the syntax definition of a data type. In particular we describe how a grammar (in SDF) can be used to generate a library of access functions that manipulate the parse trees of terms over this syntax. Application of this technique in the ASF+SDF Meta-Environment has resulted in the elimination of 47% of the handwritten code, thus greatly improving both maintainability of the tools and their flexibility with respect to changes in the parse tree format. Although the focus is on ATerms, the issues discussed and the techniques described are more generic and are relevant in related areas such as XML data-binding.

1 Introduction

Since the development of the ATerm-Library [BJKO00] in 1999, its use for the implementation of tree-like data structures has become quite popular among developers of scanners, parsers, rewrite engines and model checkers. Apart from its inevitable deployment in the tools of the ASF+SDF Meta-Environment [Kli93,BKMO97,BDH+01] for which it was specifically designed, the ATerm-Library is used amnesty others in: the ELAN system [BR01], the XT Program Transformation Tools [JVV01] which are based on the Stratego Language [Vis01a], the CoFI Algebraic Specification Language CASL [CL98,CoF98], and the μCRL ToolSet for Analyzing Algebraic Specifications [BFG+01]. ATerms include several nice features: they are easy to manipulate yet very efficient; they come with a built-in garbage collector (in the C library), and they have persistence support in the form of a compact, sharing preserving serialization in both textual and binary representations.

As more and more tools in the Asf+Sdf Meta-Environment were converted to work with the ATerm-Library, it became apparent that the tools had become inflexible with respect to changes in the parse tree format (called AsFix), and were hard to maintain. The reason behind this inflexibility was the fact that all tools used manually encoded structural knowledge about the signature of the data types, i.e. the location of data elements inside their ATerm representation. Hard-wiring such knowledge into the tools without an explicit signature definition makes it difficult, if not impossible, to change the ATerm representation of the data type.

The coding practice that uses such structural knowledge is not in any way restricted to the realm of parse trees. In fact, anyone who has programmed with the ATerm-Library, will probably be familiar with patterns such as `and(<bool>,<bool>)`. And given such a pattern, what could be easier than writing a function that extracts the arguments of the expression? But as these patterns become longer and more intricate with a liberal sprinkling of quoted strings containing backslash-escaped quotes, and as they begin to contain lists and annotations, the once so simple *make-and-match* paradigm becomes a developer's nightmare.

Shielding ATerm representation knowledge in access macros somewhat improves the legibility of code that uses them, but it does not in any way remove the maintenance issue. It restricts the knowledge to a specific set of macros, but these still need manual maintenance. As a result, this approach only *looks like* representation hiding, but in fact all programmers of different tools still need to know the exact ATerm representation of the data being exchanged.

Motivated by the need to change AsFix and to avoid the herculean maintenance task this operation would impose on our toolset, we decided to remove as much "ATerm-handicraft" from the tools as possible by developing an API-generator that creates both an interface and an implementation of data structures represented by ATerms.

While maintaining the advantages of the ATerm-Library (in our case most notably its efficiency due to *maximal subterm sharing* [1]), applications built with this generated API benefit from improved simplicity and readability, they are easier to maintain, and they are more robust against changes in the underlying AsFix representation.

This paper describes how an annotated grammar or syntax definition can be used to generate a library of functions that provide access to the parse trees of terms over this grammar. Such a library effectively turns a parse tree into

[1] 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 (sub)term sharing.

an abstract data type, providing a type-safe and systematic API to manipulate terms. In particular we describe how a SDF-specification commonly found when using the ASF+SDF Meta-Environment is used to collect the information into an *annotated data type* (ADT), necessary to build a mapping between grammar productions and their ATerm-pattern in the underlying AsFix parse tree, and how this mapping is subsequently used to generate C functions that provide an API to these parse trees. A schematic overview is shown in Figure 1.

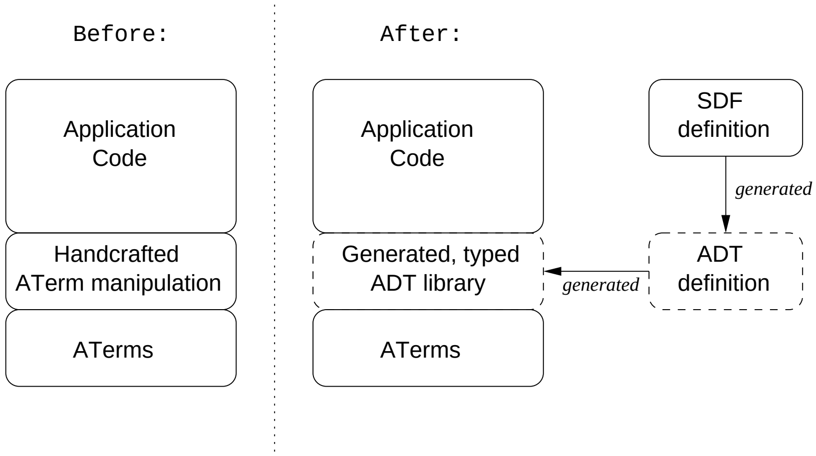

Fig. 1. Overview of an application before and after introduction of API-generation.

Although this paper uses tools from the ASF+SDF Meta-Environment as a running example, the maintenance issues addressed here are not specific to parse trees at all. The issues are fundamental to all applications that use ATerms as its data structure representation. Moreover, many of these issues are also found in applications based on other generic data representation formalisms like for instance XML.

We first relate our approach to other work in Section 1.1, and continue with some introductory sections on the specification formalism ASF+SDF (Section 1.2), the syntax of ATerms (Section 1.3), and AsFix (Section 1.4). Section 2 explains how ATerm-based data types are typically accessed in tools and applications and we show how this approach causes development and maintenance problems. We then describe the actual generation scheme from SDF to an intermediate representation (Section 3) and the subsequent generation into the target language (Section 4). The results of the application of our generation technique on tools in the ASF+SDF Meta-Environment are shown in Section 5, followed by some conclusions (Section 6), a discussion (Section 7) and future work (Section 8).

1.1 Related Work

The techniques described in this paper both use the ASF+SDF Meta-Environment and *are used* to improve the tools therein and as such our work can be seen as one step towards the implementation of [HK00]. We now briefly discuss some related work that deals with the aspects we address in this paper: applying *generational* techniques to create an *abstraction layer* on top of a *generic* data exchange formalism.

Grammars as Contracts In [JV01] a generic framework is presented that includes the generation of libraries from concrete syntax definitions. These libraries can then be used to manipulate both parse trees and abstract syntax trees. Just like our work, the instantiations are based on SDF as syntax definition formalism, in combination with tool support from the ASF+SDF Meta-Environment. Instantiations are described for generating libraries in different languages including C, Java, Stratego, and Haskell.

The work described in our paper can be seen as a refined instantiation of this generic framework. Among the instantiations described in [JV01] generation of a C library for concrete syntax manipulation is missing, and our approach remedies this situation. We also focus on generating more intuitive and readable API's, at the cost of extra annotation effort on the original syntax definition.

Zephyr ASDL

The abstract syntax definition language (ASDL) [WAKS97] is a language for describing tree data structures much like ATerms, and is used as intermediate representation language between the various phases of a compiler [Han99].

The ASDL tools support the generation of accessor and serialization code. The main differences with our approach are:

- ASDL works on abstract syntax definitions. The link between parser and ASDL must be programmed manually;
- ASDL supports a wide variety of languages including C, C++, Java, Standard ML, and Haskell;
- ASDL offers a graphical browser and editor for data described in ASDL;
- ASDL does not support maximal subterm sharing;
- There is no garbage collection support for languages like C and C++.

(D)COM/Corba IDL compiler

The two major commercial component architectures, Microsoft's (D)COM and

OMG's CORBA [Cor99,Szy97], both provide IDL compilers that take an interface definition written in their respective *interface definition languages* (IDLs), and generate communication scaffolding code. The generated code includes *stubs* and *skeletons* to make it easy to write clients and servers respectively.

The biggest difference with our work is that the target of these systems is to make it easy for programmers to build components in a distributed setting while we focus on providing an abstraction layer on top of a generic data exchange format. This means that the IDL compilers generate code for marshaling arguments when calling remote procedures and for unmarshaling their return values, while in our approach we keep the data in the original "marshaled" form until it is actually used. In this sense, our approach could be characterized as lazy and the DCOM/CORBA approach as eager.

XML data binding in Java

A comparable approach to provide an abstraction layer on top of a generic data exchange formalism is used in `jaxb` [JAX02]. This is a tool that generates a Java class hierarchy from an XML DTD. Besides accessor functions and constructors, (de)serialization functions to and from XML are generated. The actual code generation can be steered using a specification in XML. This makes it possible to add e.g. interfaces and extra code to the generated classes.

In general this approach is called *data binding*, and several other initiatives in this area are currently under way, including some open source initiatives [EP02,EG02] and the commercial initiative [BRE02]. All these approaches offer tool support for generating Java code from an XML Schema. The generated code can *marshal* and *unmarshal* XML terms to Java objects with accessors to retrieve type safe (sub)elements.

Generative Programming

Generative programming focuses on using domain engineering to retrieve domain specific knowledge that can be incorporated into component generators [CE00]. At first glance this "high level" view on generating software components seems to be far removed from the low level view on source code generation we have taken in this paper. If we take a closer look, the two approaches are not as disjoint as one might think. We believe any successful generic approach to generative programming must be based on some abstract data type definition augmented with domain specific knowledge. In our case, the data type definitions are written in SDF, and the domain specific knowledge consists of the mapping of such data types to concrete AsFix representations.

JJForester

Another approach to generating code from an SDF definition is taken in JJ-Forester [KV01]. JJForester combines the parser generator and parser from the ASF+SDF Meta-Environment with a tree builder, and visitor generator for Java. The focus lies on the generation of tree manipulation and especially tree traversal support. Especially nice is the integration with JJTraveler [Vis01b], providing generic visitor combinator support.

1.2 ASF+SDF *in a nutshell*

The specification formalism ASF+SDF [BHK89,HHKR92] is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. An overview can be found in [DHK96]. As an illustration, Figure 2 presents the definition of the Boolean data type in ASF+SDF [2]. ASF+SDF specifications consist of modules, where each module has an SDF-part (defining lexical and context-free syntax) and an ASF-part (defining equations).

1.2.1 SDF

The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation but instead arbitrary context-free grammars can be defined. SDF contains some interesting features that make it possible to give concise definitions of context-free grammars:

- Both context-free and lexical syntax can be specified.
- Lexical syntax can be described using regular expressions.
- Associativity can be specified using *attributes* (left, right, non-assoc).
- Priority relations between productions can be specified in priority sections.
- Grammar specifications can be modular.
- Modules and sorts can be parameterized.
- A number of heavily used constructs are built-in including lists, separated lists, alternatives, tuples, and function application.

The syntax defined in the SDF-part of a module can be used immediately when defining equations, thus making the syntax used in equations *user-defined.*

[2] Note how in SDF left-hand and right-hand sides of a production have opposite meaning compared to BNF notation. In SDF the elements of the LHS produce the RHS, in BNF notation the LHS is produced by the elements of the RHS.

```
module Bool
imports Layout
exports
  sorts Bool
  context-free syntax
    "true"          -> Bool
    "false"         -> Bool
    "not" Bool      -> Bool
    Bool "and" Bool -> Bool {left}
    Bool "or" Bool  -> Bool {left}

hiddens variables
    "Bool" -> Bool

equations
  [not-1] not true = false
  [not-2] not false = true

  [and-1] Bool and false = false
  [and-2] Bool and true = Bool

  [or-1] Bool or true = true
  [or-2] Bool or false = Bool
```

Fig. 2. ASF+SDF specification of the Booleans

The technology behind SDF is based on *scannerless generalized LR parsing* [BSVV02]. The term *scannerless* indicates that there is no separate scanning phase before parsing: each character is a token. This approach has the advantage that the class of languages that can be handled by the parser is not restricted by local tokenization decisions taken by the scanner.

The term *generalized* means that the parser can handle ambiguous constructs and in general yields a parse *forest* instead of a single parse tree.

To implement scannerless parsing for SDF the SDF *normalizer* is used to transform a SDF grammar into a simple character level grammar. One of the tasks of the normalizer is to explicitly insert *layout* symbols between all symbols in context-free syntax sections. For the syntax defined in Figure 2 this means that whitespace can be inserted between keywords, for instance between `not` and `true` in equation `not-1`. In this example, the actual definition of what constitutes whitespace is defined in the module Layout that is not shown in the example.

1.2.2 ASF

The equations appearing in the ASF-part of a specification have the following distinctive features:

- Conditional equations with positive and negative conditions.
- Non left-linear equations.
- List matching.
- Default equations.

It is possible to execute specifications by interpreting the equations as conditional rewrite rules. The semantics of ASF+SDF are based on innermost rewriting. Default equations are tried when all other applicable equations have failed, either because the arguments did not match or because one of the conditions failed.

The development of ASF+SDF specifications is supported by an interactive programming environment, the ASF+SDF Meta-Environment [BDH+01]. In this environment specifications can be developed and tested. It provides syntax-directed editors, a parser generator, and a rewrite engine. Given this rewrite engine terms can be reduced by interpreting the equations as rewrite rules. For instance, the term

```
true or false
```

reduces to `true` when applying the equations of Figure 2.

1.3 Annotated Terms: the ATerm syntax

The definition of the concrete syntax of ATerms is given in Appendix A. Here are a number of examples to (re-)familiarize the reader with some of the features of the textual representation of ATerms:

- Integer and real constants are written conventionally: `1`, `3.14`, and `-0.7E34` are all valid ATerms.
- Function applications are represented by a function name followed by an open parenthesis, a list of arguments separated by commas, and a closing parenthesis. When there are no arguments, the parentheses may be omitted. Examples are: `f(a,b)` and `"test!"(1,2.1,"Hello world!")`. These examples show that double quotes can be used to delimit function names that are not identifiers.
- Lists are represented by an opening square bracket, a number of list elements separated by commas and a closing square bracket: `[1,2,"abc"]`, `[]`, and `[f,g([1,2],x)]` are examples.

- A placeholder is represented by an opening angular bracket followed by a subterm and a closing angular bracket. Examples are: `<int>`, `<[3]>`, and `<f(<int>,<real>)>`.

1.4 ASF+SDF *Parse Trees for Dummies: AsFix explained*

From a SDF-specification, a parse table can be generated using the `pgen` tool from the ASF+SDF Meta-Environment. `pgen` consists of the normalizer discussed earlier combined with a parse table generator. The resulting parse table can subsequently be used by `sglr`: the scannerless, generalized LR parser to parse input terms over the syntax described by the SDF-specification. The result of a successful parse is a parse forest, containing parse trees. The data structure used to represent parse trees is called AsFix, and is implemented using the ATerm-Library to exploit the maximal subterm sharing that is commonly present in parse trees.

Because AsFix is a parse tree format (as opposed to an abstract syntax tree), layout in the input term is preserved, and other syntax-derived facts such as associativity and constructor information is made available to any tool that has access to the AsFix representation of the input term.

The definition of the concrete syntax of AsFix is given in Appendix B, but to quickly familiarize the reader with AsFix, we show some of its idiosyncrasies by means of real life examples.

Example: grammar production `"true" -> Bool`

The AsFix representation of the SDF production

```
"true" -> Bool
```

is:

```
prod([lit("true")],cf(sort("Bool")),no-attrs)
```

The `prod` symbol declares this to be a grammar production. It has three arguments: the first is a list of terminals and non-terminals that occur in the left-hand side of the production, the second argument is the non-terminal of the right-hand side, and the third argument contains the attributes (e.g. left associativity) of the production.

In this example, the literal (denoted by the symbol `lit`) `true` is the only element in the left-hand side of the production. It is injected into the context-free (denoted by the symbol `cf`) non-terminal `Bool`. The production has no specific attributes (`no-attrs`).

Example: grammar production `Bool "and" Bool -> Bool {left}`

The AsFix representation of the grammar production

```
Bool "and" Bool -> Bool {left}
```

looks like this:

```
1  prod([cf(sort("Bool")),cf(opt(layout)),lit("and"),
2        cf(opt(layout)),cf(sort("Bool"))],
3    cf(sort("Bool")),
4    attrs([assoc("left")]))
```

- Lines 1 and 2 declare this to be a grammar production (`prod`), containing all the elements of the left-hand side of the production. The SDF-normalizer has inserted the context-free sort `opt(layout)` subterms at every location where optional layout in the input term is allowed.
- Line 3 tells us that the result sort of this production is `Bool`.
- Line 4 shows the attributes associated with this production. In this case the only attribute is `left` for left-associativity.

Example: parsed term `true and false`

If the input term

```
true and false
```

is parsed, the resulting parse tree is the production from the previous example, applied to the actual argument `true and false`. The layout in the input term consists of exactly one space immediately before and after the keyword `and`.

```
1 appl(
2  prod([cf(sort("Bool")),cf(opt(layout)),lit("and"),cf(opt(layout)),
3        cf(sort("Bool"))],cf(sort("Bool")),attrs([assoc("left")])),
4 [appl(prod([lit("true")],cf(sort("Bool")),no-attrs),[lit("true")]),
5  layout([" "]), lit("and"), layout([" "]),
6  appl(prod([lit("false")],cf(sort("Bool")),no-attrs),
7       [lit("false")])])
```

- Line 1 states that this tree is the application of a grammar production to a specific term.

- Lines 2–3 show the representation of `Bool "and" Bool -> Bool {left}` from the previous example.
- Line 4 shows the application of the production `"true" -> Bool` to the literal `true`.
- Line 5 contains the instantiated optional layout terms. In this case the input term contained exactly one space immediately before and just after the keyword `and`.
- Similar to line 4, lines 6–7 represent the literal `"false"`.

The fact that many tools in the ASF+SDF Meta-Environment need to operate on such parse trees, raises the question of how best to access this ATerm representation of a data type.

2 Accessing ATerm Data Types

The ATerm-Library provides two levels of access to ATerms. We briefly discuss both of them (Sections 2.1 and 2.2) by showing some examples using the C implementation of the ATerm-Library. Similar statements are needed when using the JAVA implementation.

Section 2.3 shows the typical way tools in the ASF+SDF Meta-Environment used to access AsFix parse trees. As AsFix terms are of impressive complexity to the human eye, the code needed to access them becomes equally complex if it has to be written down manually.

2.1 Accessing ATerms using the Level One interface

The first level of access functions is through the easy-to-learn *make and match* paradigm which allows construction of terms by parsing their string representation. Placeholders in these patterns are used to designate "holes" in the term which are to be filled in by other variables, including other ATerms as well as native types (`int`, `string`, etc.). Terms are constructed using `ATmake`, for example:

```
ATerm t = ATmake("person(name(<str>),age(<int>))", "Anthony", 7);
```

will result in term `t` being assigned the value:

```
person(name("Anthony"),age(7))
```

Note how the placeholders <str> and <int> are substituted by the values Anthony and 7, respectively.

Elements from terms can be extracted using ATmatch, for example:

```
char *name;
int age;
if (ATmatch(t, "person(name(<str>),age(<int>))", &name, &age)) {
  printf("name = %s, age = %d\n", name, age);
}
```

will result in the variables name and age being assigned the values Anthony and 7, respectively. The output of this fragment would thus be:

```
name = Anthony, age = 7
```

In case we are only interested in extracting the age field and we do not care about the actual value of name, we can pass NULL instead of the address of a local variable. In this case, that particular subterm is still used during matching, but its actual value is never assigned. This allows us to test if a specific term matches a given pattern, without having to bind every placeholder in the pattern.

2.2 Accessing ATerms using the Level Two interface

The second level of access allows more direct manipulation of ATerms by means of access-functions which operate directly on a term or its subterms. This way of access is more efficient than using the level one interface, because there is no need to parse a string pattern to find out which part of the (sub-)term is needed.

For example, consider the term from the previous section:

```
t = person(name("Anthony"),age(7))
```

We can get Anthony's age by first extracting the age subterm from t, and subsequently getting the actual 7 from this age term. Arguments in an ATerm function application are numbered, starting at zero. So, to get to the actual value of 7 which is embedded in the age function application, we need to extract argument number 1 from the person application, and then extract argument number 0 from this:

```
int age = ATgetInt(ATgetArgument(ATgetArgument(t, 1), 0));
```

Note that the exact *location* of the age field in the ATerm representation of the `person` record is used. If the structure of the record were to change, e.g. a field for the person's last name is inserted between the `name` and the `age` fields, the example code would be broken.

Also note that this code does not even check if the term `t` is of the right form, i.e. if `t` satisfies the pattern `person(name(<str>),age(<int>))`. On an arbitrary input term, the age-extraction code will most likely fail and dump core. But if only correct input terms are given, it is the most efficient way to encode the extraction of the age subterm in this ATerm representation of the `person` record.

2.3 Accessing AsFix parse trees

This Section shows several ways in which AsFix terms can be accessed. The code fragments are typical for the way parse trees are manipulated in the ASF+SDF Meta-Environment.

First, we show the C code necessary to construct the boolean term `true` which, when yielded by the parser, looks like this:

```
appl(prod([lit("true")],cf(sort("Bool")),no-attrs), [lit("true")])
```

Even for such a simple input term, its ATerm representation written as a C (or JAVA) string is already quite complex. This is because we have to escape all the double quotes (the `"` characters) from interpretation by the compiler. Also, because the string representation of the match-pattern is long enough that it does not legibly fit on a single line anymore, we have to resort to ANSI C string concatenation[3] to span the string over multiple lines.

```
ATerm true = ATparse(
  "appl(prod([lit(\"true\")],cf(sort(\"Bool\")),no-attrs),"
  "[lit(\"true\")])");
```

As another example, consider a C function that extracts the left-hand side from a boolean conjunction. It needs to match the parse tree of the incoming term against the pattern for the syntax production:

```
Bool "and" Bool -> Bool {left}
```

An implementation using the level one interface would need the pattern written as a string, with a `<term>` placeholder at the correct spot. Because the pattern is written inside a string, we once again need to escape all quotes.

[3] Strings can be split over multiple lines by ending one line with a `"` and starting the next line with another `"`.

```
ATerm extract_bool_lhs(ATerm t) {
  ATerm lhs;
  char *bool_and_lhs_pattern =
    "appl(prod([cf(sort(\"Bool\")),cf(opt(layout)),"
    "lit(\"and\"),cf(opt(layout)),cf(sort(\"Bool\"))],"
    "cf(sort(\"Bool\")),attrs([assoc(\"left\")])),"
    "[<term>,<term>,lit("and"),<term>,<term>])";

  if (ATmatch(t, bool_and_lhs_pattern, &lhs, NULL, NULL, NULL)) {
    return lhs;
  }

  return NULL;
}
```

Could there be a quote missing in the pattern? Are all the `)`, `]`, and `}` characters where they should be? Did you expect four `<term>` placeholders in the pattern (to account for the lhs, the rhs, as well as the optional layout before and after the literal `and`)?

Keep in mind that:

- as long as it is a valid C string, the C compiler is not going to warn you if you make a mistake (e.g. you wrote `lit(and)` instead of `lit(\"and\")`);
- as long as it is a valid ATerm-pattern, the ATerm parser is not going to warn you if you make a mistake (e.g. you forgot to add `<term>` placeholders for the optional layout);
- if you made any mistakes, your only hope to fix them lies in visually inspecting the incoming term and the expected matching pattern, and figuring out why they do not match!

An implementation using the level two interface encodes structural knowledge about the exact location of the `lhs` in terms of direct ATerm access functions. In particular, recalling that in AsFix we are dealing with `appl(prod,[args])` patterns, the `args` are always the second argument of the `appl`. If we look closely at the AsFix pattern for our `and`-terms, we notice that the `lhs` is the first element in this list of `args`. The extraction function can thus be simplified to the more efficient, but very type-unsafe and obfuscated:

```
ATerm extract_bool_lhs(ATerm t) {
  /* get arguments from AsFix "appl" */
  ATermList args = ATgetArgument(t, 1);

  /* lhs is the first of these args. */
  return ATgetFirst(args);
}
```

After all, this function would work on any ATerm function application that has (at least) two arguments, the first of which is a list with (at least) one element.

2.4 Maintenance issues

There are several fundamental maintenance issues inherent in the use of ATerms as a data structure implementation in hand-crafted tools.

- The esoteric art of writing down multi-line, quote-escaped string patterns and the subsequent substitution of parts of these patterns to contain the desired placeholders at the correct locations, is so error prone that it is almost guaranteed to go wrong at some point. Practical experience in the ASF+SDF Meta-Environment has proven this many times over. Handcrafted ATerm-patterns proliferate through numerous versions of various tools, and after a while all sorts of "mysterious" bugs creep up where one tool cannot handle the output of another tool, or simply bails out reporting that deep down some part of an input term does not satisfy a particular assertion. Obviously, these errors are often due to pattern mismatches, misplaced placeholders, or ill-escaped quotes.
- Even if the patterns are written down correctly, or when the Level Two interface is used (which doesn't use ATerm-patterns), there is much work to be done when the application syntax changes.

  Suppose for example that we want to change the syntax of our boolean conjunction from infix notation:
  ```
  Bool "and" Bool -> Bool
  ```
  into prefix notation:
  ```
  "and" "(" Bool "," Bool ")" -> Bool
  ```
  Conceptually nothing has changed: we mean exactly the same arguments when we address them as `lhs`, `rhs`, and `result` terms in both productions. However, in the underlying parse tree the location of *all three* subterms has changed! This in turn means that all tools that manipulate, e.g. the `lhs` of boolean terms, have to be updated to reflect this structural change.

  In fact, there is hardly any room for flexibility with respect to changes in the syntax, unless the arguments happen to remain at their original position. Every tool based on the modified *application syntax* has to be updated.
- With such inflexibility with respect to the application syntax in mind, imagine what would happen if the structure of the parse trees (AsFix) *itself* were to change. Every tool based on the *representation* of parse trees would have to be updated to reflect the structural changes in the format. In our practical case of the ASF+SDF Meta-Environment where we wanted to rid AsFix of some legacy constructs, this meant modification of virtually *every* tool — an arduous task indeed.

3 From syntax to API

Abstracting from implementation details about the facts that there is such a thing as a parse tree format and that this format in turn is implemented using ATerms, it is easy to name several operations a tool-builder would like, given a syntax definition.

As an example we consider the booleans again. Some of the typical things a tool-builder would like to be able to do given the boolean syntax are:

- Use a type definition for booleans (it is better to have a specific type `Bool` than to use the generic `ATerm` type);
- Create the basic booleans: `true` and `false`;
- Create a compound boolean term using basic and other compound boolean terms;
- Given an arbitrary term, test if it is a valid boolean term;
- Given an arbitrary boolean term, distinguish between a basic term and a compound term, e.g. by testing if it has a `lhs` or `rhs`;
- Extract the `lhs` and `rhs` of a given boolean term;
- Replace the `lhs` and `rhs` of a compound boolean term by another boolean term;
- etc.

Obviously, this list is not exhaustive, but it does form a nice starting point. Fortunately, all the necessary information can be extracted from an SDF-definition of the grammar. In order to separate some concerns and simplify the generation framework, we split the process into two steps (see Figure 3). First, we extract all the necessary information from the SDF-definition, and store it in a convenient format. This step takes care of the parsing and analysis of the grammar. The second step takes the intermediate format and does the actual generation for a specific target language.

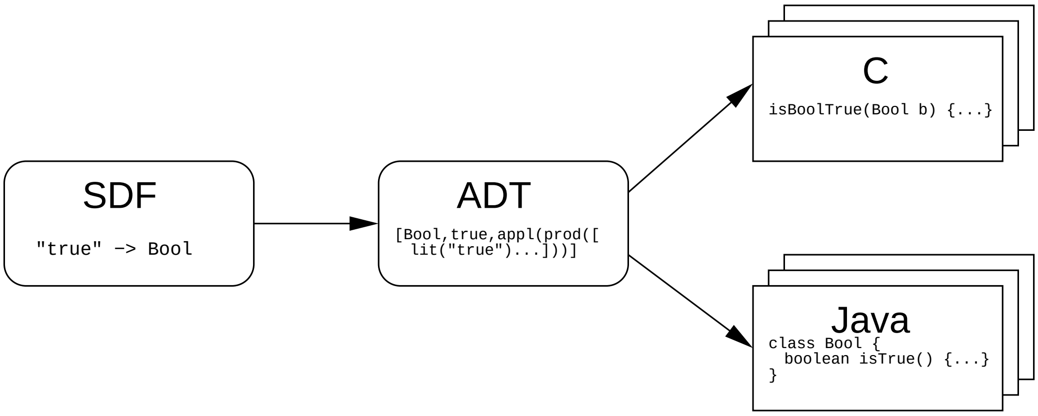

Fig. 3. Generation scheme: from SDF to ADT to code

We call the intermediate format *annotated data type*, or ADT for short.

It holds the minimal amount of information for each syntax rule in the original SDF specification. In particular, for each rule we need:

- The *sortname* of the production. In our boolean syntax this is `Bool`;
- The *alternative* of the production. Our boolean syntax (from Figure 2) has five alternatives: `true`, `false`, `not`, `and`, `or`.
- The actual ATerm-*pattern* representation of the rule. In this pattern, each *field* (non-terminal in the syntax rule) is replaced by a typed placeholder containing the *sort* of the non-terminal and a *descriptive name*. For the `and` rule we could use `lhs`, and `rhs`, both of type `Bool`.

Since we are solving the maintenance problem of using ATerms as a data type representation, we decided we could very well use an ATerm to represent the elements of an ADT. The obvious advantage is that we get persistence (saving and loading of an ADT) for free, and we do not need to construct a domain specific language (with its own parser etc.) which would introduce undesired development-time overhead. Each entry in the ADT consists of the three elements *sortname*, *alternative*, and *term-pattern*, which we can easily represent as an ATerm-list. An entire ADT consists of nothing more than a list of such lists. Instead of using a list, each single entry could also have been represented as a function with three arguments, but we opted for as little syntactic sugar in the entries as possible, to simplify development-time debugging. Remember that an ADT entry contains an ATerm pattern and they are hard enough to read, without the introduction of an extra function-symbol around them.

As an example of the concrete representation of an ADT entry, let us look at the boolean `and`. In this example, we know that the sortname of the production is `Bool`, the alternative is called `and`. There are two operands, `lhs` and `rhs`, both of type `Bool`. In the pattern we put typed placeholders `<lhs(Bool)>` and `<rhs(Bool)>` at the location of the corresponding non-terminals. Also, because this is a parse tree pattern, we have to allow layout (whitespace), which in this case can occur both after the non-terminal `lhs`, and after the literal `and`. The ADT entry thus becomes:

```
1  [Bool,
2   and,
3   appl(prod(
4     [cf(sort("Bool")),cf(opt(layout)),lit("and"),cf(opt(layout)),
5      cf(sort("Bool"))],cf(sort("Bool")),attrs([assoc(left)])),
6     [<lhs(Bool)>,<ws-after-lhs(Layout)>,lit("and"),
7      <ws-after-and(Layout)>,<rhs(Bool)>])]
```

- Line 1 contains the sortname: `Bool`

- Line 2 shows the alternative: `and`
- Lines 3–5 show the `prod` of the AsFix function application.
- Lines 6–7 show the `args` part. Clearly visible are the typed placeholders for `lhs` and `rhs`.

The two placeholders matching optional layout have the somewhat arbitrary names `ws-after-lhs`, and `ws-after-and`. Section 3.1 elaborates on the naming schemes used to generate legible, understandable names.

Given an ADT, which is generated from an SDF definition, but which could also come from any other source, we no longer need to worry about any SDF peculiarities, or parse tree specifics. Instead, we can concentrate on generating the desired functionality for a given target language. In this paper we concentrate on describing the steps needed to produce legible, type-safe C code. Optimizations to the generated code can easily be obtained by removing type-safety checks, resulting in a more efficient production version of the code.

3.1 Deriving the ADT *from a* SDF *specification*

Now that we know what specific information we need in the ADT, how do we get it from the SDF definition? If we look back at our SDF definition of the booleans, we can derive two of the necessary elements immediately:

- The result *sort* of a syntax rule. It is explicitly mentioned at the end of each rule.
- The ATerm pattern. It can be constructed by following the exact same rules for constructing AsFix terms that the SDF normalizer uses.

This leaves us with the issue of coming up with a decent name for each *alternative* production of the same sort, and we still need to figure out a way to give *descriptive* names to the non-terminals in the grammar rule.

Naming the non-terminals

Given our SDF rule for the boolean `and`, can we derive a sensible name for each of the `Bool` non-terminals? The only information we have is our syntax rule:

```
Bool "and" Bool -> Bool {left}
```

If we use heuristics to call them e.g. `lhs` and `rhs`, what do we do when we find another syntax rule that has three, four or even more arguments? In syntax rules with only one non-terminal, we could default to using the sort name of that non-terminal. But in general, it is hard to come up with any kind of descriptive naming scheme. Keep in mind that most tool-builders will not really be happy if they are confronted with access functions that have arbitrarily complex names, or numbered arguments.

Instead of coming up with any kind of heuristic at all, we opted to use the *labeling* mechanism present in SDF, which allows grammar writers to label each non-terminal. This eliminates the need to invent a descriptive name altogether and provides an understandable link between items in a grammar rule and their generated access functions. Suppose we like the abbreviations `lhs`, and `rhs`, we could label the syntax rule for `and` to become:

```
lhs:Bool "and" rhs:Bool -> Bool {left}
```

Naming the alternatives

Similarly, we need a solution for the *alternative* name. In this case the literal `and` happens to be a name we could use. But what if there is no literal at all? Or if there are multiple literals in a production, which one should we pick? Should they be concatenated? What if the literal is some sort of baroque lexical expression (think of the C and JAVA symbols `&&` for conjunction). Again we are saved by SDF, which provides a way to annotate syntax rules. In fact, we re-use an annotation which is quite commonly used by SDF syntax writers to annotate the name of the *abstract syntax* node that corresponds to this particular syntax rule. Traditionally the `cons` annotation is used for this purpose. So, finally our `and` syntax rule becomes:

```
lhs:Bool "and" rhs:Bool -> Bool {left, cons("and")}
```

From which we can subsequently generate (e.g. C) type and function names as shown in Figure 4.

4 Code generation from ADT to C

4.1 Generated types and functions

For each sortname in an ADT, we generate the following items (which are further explained in Subsection 4.2):

- An opaque type definition to distinguish instances of this particular sort from other ATerms.
- Conversion functions `fromTerm` and `toTerm` to interface with generic ATerm functions, such as `ATreadFromFile`. These functions perform a type cast, and as such they form the entry and exit points to type-safety.

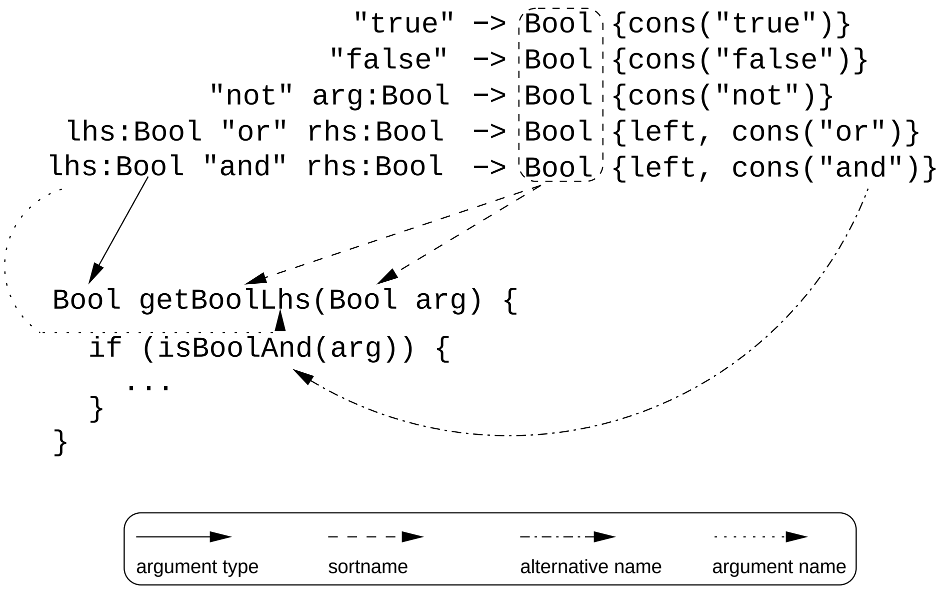

Fig. 4. Using SDF elements to derive legible names.

- A validity function to test whether an instance of a sort is indeed valid, i.e. that it indeed matches one of the ATerm-patterns defined as an alternative of this sort. This is useful to assert the validity of an externally acquired instance of this sort, e.g. if it has just been read from file.
- Constructor functions for each possible alternative for this sort to create instances from scratch.
- An equality function to test equality with another instance of this sort.
- For each alternative of the sort, an `isAlternative` function that checks if the current object is an instance of that particular alternative.
- For each field used in any of the alternatives of the sort, a `hasField` function that checks if the current object is an instance of an alternative that has that non-terminal.
- Similarly, a `getField` and `setField` method for each of the fields in a sort.

4.2 Implementation

In order to address the maintenance issues associated with the proliferation of ATerm-patterns, we decided it was a good idea to isolate them as much as possible from the actual code. This is achieved by generating a separate *dictionary* file containing all the ATerm patterns used by the library. This *dictionary* file declares a separate `AFun` variable for each ATerm function symbol, and an ATerm variable for each possible pattern. An initialization function is also generated which takes care of the proper initialization of all these variables, and the necessary calls to `ATprotect` to shield them from the built-in garbage collector. A verbatim dump of all the patterns is included in a comment section in the generated code, to provide debugging feedback when necessary. An example of a dictionary file can be found in Appendix C.

The actual implementation of the API functions is generated in its own C file, accompanied by a header file containing the signatures of all exported API functions. We show abridged snippets of the generated code. The header file is straightforward, containing merely the opaque type definition, and the declarations of the functions contained in the C file.

Opaque type definition

Defining `Bool` to be a pointer to a non-existent type (in this case `struct _Bool`, hides the underlying ATerm representation from the point of view of API users. Instances of `Bool` can safely be passed around by functions, but any attempt to dereference such a pointer results in a compile time error.

```
typedef struct _Bool *Bool;
```

Term convertors

These functions perform no real operation, but take care of the type casting between the generic ATerm type and the more specific `Bool`. They are needed as entry and exit points to type-safety when ATerm-Library functions such as `ATreadFromFile` are used, which yield an ATerm.

```
Bool BoolFromTerm(ATerm t) { return (Bool)t; }
ATerm BoolToTerm(Bool arg) { return (ATerm)arg; }
```

For improved efficiency, these functions could easily be replaced by macros which perform the exact same type cast. Unfortunately, this irrevocably kills type-safety, because macros are expanded during the pre-processor phase, without any form of type checking on the arguments of the macro.

Equality test

Because ATerms are used as implementation, we get the trivial equality check based on memory address comparison for free. We only need to provide a type-safe wrapper around `ATisEqual`.

```
ATbool isEqualBool(Bool arg0, Bool arg1) {
  return ATisEqual((ATerm)arg0, (ATerm)arg1);
}
```

As with the convertor functions, the equality function can be replaced by a macro definition (with the same concerns about the loss of type-safety) for improved efficiency.

Validity test

Whenever an ATerm is acquired from an external source (e.g. by reading it from file) and is converted to `Bool`, programmers might like to assert that the term satisfies one of the alternatives for `Bool`. After all, any valid ATerm will happily be parsed by `ATreadFromFile` and subsequent conversion by `TermToBool` is done without any verification. The `isValidBool` function checks whether a given `Bool` argument is indeed an instance of one of the correct alternatives.

```
  ATbool isValidBool(Bool arg) {
    if (isBoolTrue(arg)) { return ATtrue; }
    else if (isBoolFalse(arg)) { return ATtrue; }
    else if (isBoolNot(arg)) { return ATtrue; }
    else if (isBoolAnd(arg)) { return ATtrue; }
    else if (isBoolOr(arg)) { return ATtrue; }
    return ATfalse;
}
```

As checking the alternatives is expensive, the conversion functions themselves do not directly invoke `isValidBool`. Efficiency of the `BoolToTerm` function could be traded for even more robustness, by making it refuse to convert any ATerm that does not satisfy `isValidBool`.

Also note that the `isBoolX` functions perform a *shallow* match: they do not check the *arguments* of the alternative they test. For example, `isBoolAnd` does not check if its `lhs` and `rhs` are actually valid booleans. It merely tests if the term is an instance of the pattern for the `and` alternative. It would be possible to generate code that performs a *deep* match, again at the cost of a considerable efficiency hit.

Inspector

Inspecting a `Bool` to see if it is an instance of a specific alternative involves matching the argument against the pattern for that particular alternative. Because matching is expensive, the result of the most recent match is cached. This caching approach seems limited, but is useful when multiple subterms of the *same argument* are accessed. In these cases, sequences of `getBoolX` and `setBoolY` all reuse (cached) inspection results.

```
  ATbool isBoolTrue(Bool arg) {
    static ATerm cached_arg = NULL;
    static int last_gc = -1;
    static ATbool cached_result;

    assert(arg != NULL);
```

```
    if (last_gc != ATgetGCCount() || (ATerm)arg != cached_arg) {
      cached_arg = (ATerm)arg;
      cached_result = ATmatchTerm((ATerm)arg, patternBoolTrue);
      last_gc = ATgetGCCount();
    }

    return cached_result;
  }
```

Note that the cached ATerm is deliberately *not* protected from garbage collection. Doing so would result in all inspector functions holding on to references of ATerms that could not be collected. These terms are potentially very large and the memory behaviour would become extremely unpredictable. We therefore opted for a solution where caching results are only valid *until the next garbage collection*. This is done by comparing the current garbage collection count with the same count at the time the cached result was calculated.

Query accessor

The query accessor checks if a given argument has a specific field. It is implemented by checking if the argument is an instance of any of the alternatives which has the required field.

```
  ATbool hasBoolLhs(Bool arg) {
    if (isBoolAnd(arg)) { return ATtrue; }
    else if (isBoolOr(arg)) { return ATtrue; }
    return ATfalse;
  }
```

Get accessor

The `get`ter is implemented much like the query accessor. It inspects the incoming argument to find out of which alternative it is an instance. Upon finding the right alternative, it returns the intended subterm by directly peeking into the ATerm representation.

If the production has but a single alternative, no testing is needed and the requested subterm can be returned immediately.

```
  Bool getBoolArg(Bool arg) {
    return (Bool)
      ATelementAt((ATermList)ATgetArgument((ATermAppl)arg, 1), 2);
  }
```

If there are multiple alternatives, each is tested in turn, until a single alternative remains, which must be the right one (since none of the other alternatives matched, and we assume a valid instance of one of the alternative productions).

```
  Bool getBoolLhs(Bool arg) {
    if (isBoolAnd(arg)) {
      return (Bool)
        ATgetFirst((ATermList)ATgetArgument((ATermAppl)arg, 1));
    }
    else
      return (Bool)
        ATgetFirst((ATermList)ATgetArgument((ATermAppl)arg, 1));
  }
```

An obvious optimization would be to detect if there are alternatives that have the requested field at the same location in the underlying ATerm representation. In this case, the `isBoolAnd` test is redundant, because both alternatives of `Bool` that have a `lhs`, have it at the exact same position. The condensed version would look much like the previous `getBoolArg` and would be much cheaper since it does not have to do any matching:

```
Condensed code for alternatives: "and", "or"
  Bool getBoolLhs(Bool arg) {
    return (Bool)
      ATgetFirst((ATermList)ATgetArgument((ATermAppl)arg, 1));
  }
```

Set accessor

The implementation of the `set`ter is again along the same path as the `get`ter and the inspector. The main issue here stems from the fact that ATerms are immutable. Consequently, all `set`ters need to be of a functional nature. This means that they cannot update the ATerm *in situ*, but instead need to construct a *new* ATerm, reflecting the desired change.

```
  Bool setBoolLhs(Bool arg, Bool lhs) {
    if (isBoolAnd(arg)) {
      return (Bool)
        ATsetArgument((ATermAppl)arg, (ATerm)
                      ATreplace((ATermList)
                                ATgetArgument((ATermAppl)arg, 1),
                                (ATerm)lhs, 0), 1);
    }
    else if (isBoolOr(arg)) {
      return (Bool)
        ATsetArgument((ATermAppl)arg, (ATerm)
                     ATreplace((ATermList)
                               ATgetArgument((ATermAppl)arg, 1),
                               (ATerm)lhs, 0), 1);
```

```
  }

  ATabort("Bool has no Lhs: %t\n", arg);
  return (Bool)NULL;
}
```

As the construction of a new ATerm is expensive to begin with, the gain of eliminating the test for one of the alternatives (as implemented in the `get`ters) is minimal, which is why that particular optimization is omitted here. If no match was found after exhaustively testing all the alternatives, the operation is aborted.

5 Software engineering benefits in the ASF+SDF Meta-Environment

Our main motivation to start this work has been the desire to make changes to AsFix, the parse tree format used by our tools in the ASF+SDF Meta-Environment. Of particular interest is the dramatic size reduction (in terms of lines of code) of the various tools after refactoring them to use the new APIs.

This *apification* process consisted of the following stages:

- Reverse engineering the actual interfaces (and the corresponding term representations) that were needed in the ASF+SDF Meta-Environment. This resulted in three ADT's:
  - · A handwritten ADT for our parse tree format AsFix, closely matching the structure of the parse trees as they were produced by our parser;
  - · An ADT for SDF, generated from our SDF definition of SDF;
  - · An ADT for ASF, generated from a SDF definition of ASF.

  As a result, we ended up with a specification for the main three formats used in the ASF+SDF Meta-Environment. The main purpose of these specifications is to provide an authoritative description of the formats used, and to generate a consistent API as described in this paper.
- Replacing all direct, untyped ATerm-manipulations by typed calls to the generated API. In practice, this often amounted to replacing large sections of code by concise snippets or even a single invocation of the API, clearly demonstrating the transition to a higher level of abstraction. The fact that we now operate in a typed domain means that we are able to effectively track type-related problems using the C compiler. We were even able to locate and fix a number of severe bugs in the *original* code that had not yet manifested themselves.

After the apification process was complete, we achieved a significantly higher level of maintainability of the code. We are now able to implement changes in the term representation, which was one of our major goals. Moreover the higher level of abstraction allows us to implement new functionality which would otherwise be much more time consuming and error prone. For example, it allowed us to quickly write an ASF-checker which traverses ASF-equations looking for occurences of uninstantiated variables.

In accordance with these subjective observations that the code has improved, is the Lines of Code (LOC) metric. Comparing versions just before and immediately after *apification*, we found out that we had been able to eliminate almost half of the (manually written) code. The LOC metrics have been summarized in Figure 5.

The components are: the runtime environment of the ASF+SDF compiler (`asc-runtime`), the parse tree library (`libasfix`) and utilities (`asfix-tools`), the actual ASF+SDF compiler (`asf+sdf compiler`), a collection of ASF manipulation utilities (`asf-tools`), a `structure editor` for editing ASF+SDF specifications, an `evaluator` for evaluating (rather than compiling) ASF equations, and finally a repository for parse tables and parsed ASF+SDF specifications (`module-db`).

Component	Before (LOC)	After (LOC)	Reduction (%)
asc-runtime	2207	1752	21
libasfix	10419	2077	80
asfix-tools	466	603	-29
asf+sdf compiler	1866	1138	39
asf-tools	1303	589	55
structure editor	2861	1946	32
evaluator	4241	4009	5
module-db	1809	1244	31
Total	**25172**	**13358**	**47**

Fig. 5. Code Reduction

Understandably the biggest gain was achieved in `libasfix`, because most of this library is now generated from the SDF definition of SDF itself. Only some high level functionality that could not be generated remains in this library.

6 Conclusions

Generating access libraries from SDF definitions offers a simple, consistent way of developing and maintaining type-safe, efficient data types.

The application in the ASF+SDF Meta-Environment of the techniques described in this paper resulted in the elimination of a significant portion of handcrafted code. The effect of this elimination is amplified by the inherent nature of the affected code portions: hard to read and write, difficult to maintain, and in general very error prone to handle at all.

The result of our generational approach is a type-safe replacement for manually crafted libraries that provide access to compound data types implemented by ATerms. Even though several optimization opportunities have not yet been fully explored, the efficiency of the generated library is already comparable to its manually written predecessor.

More generically speaking, the approach presented in this paper is applicable in situations where type safety is needed at a different (higher) abstraction level than is offered by the underlying data format. This is especially true in situations where representing the data at the higher abstraction level directly is unfeasible, e.g. due to performance issues.

7 Discussion

Although this paper shows how API generation was used in a very specific context (generating ATerm manipulation code from an SDF specification), many of the issues encountered are not ATerm or SDF specific at all. In fact any generic data exchange formalism potentially suffers from the problem that generic manipulation functions are inherently type unsafe. For instance if we look at XML, DOM based libraries for manipulating XML in a generic way suffer from many of the the same problems as the ATerm library. Recent techniques like XML data binding (discussed in Section 1.1) take the same approach as we do in this paper by generating accessor and manipulation functions based on a signature description like XML schemas or DTDs.

In retrospect, one might ask why we ever developed code using direct ATerm manipulations in the first place. The answer lies partially in the power and attractiveness of working with ATerms. Because it is so easy to write a tool that uses simple ATerm patterns, several developers quickly started writing their own applications. Later, when some of the tools demonstrated a need for speed, parts of the now grown-but-not-restructured tools were rewritten to use the more efficient level two interface instead of the matching interface, mostly in the form of ad hoc restructuring, driven by the output of the `gprof` profiler. When prompted to implement changes in our parse tree format, we realized the era of direct ATerm manipulation had to end, and we had to find a structural solution to representing data types by ATerms, or we would be unable to maintain our toolset. Fortunately, the road of generating the access library as described in this paper works very well in the ASF+SDF Meta-Environment. Since the introduction of what has become known as APIGEN, we have been able to effectuate considerable changes in AsFix, and we have gained the ability to experiment with the format, and quickly see the results working in our tools.

8 Future work

The future work of this project falls into two categories: increasing the efficiency of the generated code and generalizing our approach to a wider application area.

Obvious optimizations include *inlining* (some of) the generated functions. By rewriting the functions as C macros, the overhead of a function call is removed. As noted before, the cost of this efficiency gain is that some type-safety is lost. This is due to the fact that C macro expansion is performed by a pre-compiler, which does not have access to type information and thus performs no type checks on macro arguments. A typical approach would be to generate type-safe functions in the development stage, and switch to the use of generated efficient macros for production code. This approach is comparable to the use of `assert` macro's that are completely eliminated in production versions of the software.

More interesting, however, are optimizations that take into account information about the structure of the underlying ATerm representation. During the generation phase information about common subterms and similarity between alternatives is assembled, which could be exploited to generate more efficient matching and selection code than the current `ATmatch` call, which is rather inefficient.

In a way the project described in this paper can be seen as a case study in generative programming as described in Section 1.1. We want to extend this case study into a more generic approach. This approach will be based on a modular generic generator that takes a set of abstract data definitions and generates code for them. The code generator must be extensible with domain specific "modules" to generate extra functionality. These modules should not only be able to add extra functions when needed, but they should also be able to use *Aspect Oriented Programming* [KLM+97] to add functionality to functions that are generated by other modules.

For example, one of the more basic modules (the "accessor" module) could generate the actual data representation (for instance simple attributes in an object oriented setting) and accessors on this representation, while another module could add transparent persistency using a standard relational database by instrumenting all accessors.

The most important challenge in such an approach would be to create an environment where the threshold to create new generator modules is extremely low. In the ideal situation software developers could add new modules to the generator just as easy as to add new modules directly to the software system they are building.

Availability

Users interested in the more technical details (i.e. the actual implementation) or who would like to deploy the tools we described, are encouraged to download the latest distribution from:

`http://www.cwi.nl/projects/MetaEnv/apigen`

References

[BDH+01] 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, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[BFG+01] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer, July 2001.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[BJKO00] 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.

[BKMO97] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and Implementation of a New Asf+Sdf Meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, November 1997. Springer-Verlag.

[BR01] M.G.J. van den Brand and C. Ringeissen. ASF+SDF parsing tools applied to ELAN. In Kokichi Futatsugi, editor, *Third International Workshop on Rewriting Logic and its Applications (WRLA'2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[BRE02] XML/Java Data Binding and Breeze XML Binder. Technical report, The Breeze Factor, 2002. available by `http`[4].

[BSVV02] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR parsers. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

[CE00] K. Czarnecki and U.W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CL98] CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [CoF98], 1998.

[CoF98] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents available by `http`[5], 1998.

[Cor99] *The Common Object Request Broker: Architecture and Specification*. Object Managament Group (OMG), 1999.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[EG02] The ExoLab Group. Castor - a mapping framework between Java objects, XML documents, SQL & OQL databases and LDAP directories. Available via `http`[6], 2002.

[EP02] The Enhydra Project. The Zeus Java-to-XML Data Binding tool. Available via `http`[7], 2002.

[Han99] D.R. Hanson. Early Experience with ASDL in lcc. *Software - Practice and Experience*, 29(3):417–435, 1999.

[4] `http://www.breezefactor.com/whitepapers.html`
[5] `http://www.brics.dk/Projects/CoFI/`
[6] `http://castor.exolab.org`
[7] `http://zeus.enhydra.org`

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual*, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989.

[HK00] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, March 2000.

[JAX02] Java architecture for XML binding. Technical report, SUN Microsystems, 2002. available at: `http://java.sun.com/xml/jaxb`.

[JV01] M. de Jonge and J. Visser. Grammars as contracts. In Greg Butler and Stan Jarzabek, editors, *Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000*, volume 2177 of *Lecture Notes in Computer Science*, Erfurt, Germany, 2001. Springer.

[JVV01] M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In Mark van den Brand and Didier Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[KV01] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA).

[Szy97] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[Vis01a] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[Vis01b] J. Visser. Visitor combination and traversal control. *ACM SIGPLAN Notices*, 36(11):270–282, November 2001. OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.

[WAKS97] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

A Concrete Syntax of ATerms

A formal definition of the concrete syntax for ATerms using SDF is presented here.

```
module ATerms

hiddens
  sorts EscChar AFunChar Annotation
  lexical syntax
  "\\" ~[]                          -> EscChar
  "\\" [01][0-7][0-7]               -> EscChar
  ~[\000-\040\"\\]                  -> AFunChar
  EscChar                           -> AFunChar
  "{" ATerms "}"                    -> Annotation

exports
  sorts ATermInt ATermReal AFun
  lexical syntax
    [0-9]+                          -> ATermInt
    "-" [0-9]+                      -> ATermInt
    ATermInt "." [0-9]+             -> ATermReal
    ATermInt "." [0-9]+ [eE] ATermInt -> ATermReal

    [a-zA-Z][a-zA-Z0-9\_\-]*        -> AFun
    "\"" AFunChar* "\""             -> AFun

  sorts ATermAppl ATermPlaceholder ATermList ATerm
  context-free syntax
    AFun                            -> ATermAppl
    AFun "(" ATerms ")"             -> ATermAppl

    "<" ATerm ">"                   -> ATermPlaceholder
    "[" "]"                         -> ATermList
    "[" ATerms "]"                  -> ATermList

    ATermInt Annotation?            -> ATerm
    ATermReal Annotation?           -> ATerm
    ATermAppl Annotation?           -> ATerm
    ATermList Annotation?           -> ATerm
    ATermPlaceholder Annotation?    -> ATerm
```

```
  aliases
    {ATerm ","}+                           -> ATerms
```

B Concrete Syntax of AsFix

A formal definition of the concrete syntax for AsFix using SDF is presented here.

```
module Symbol
imports Literals

exports
  sorts Symbol Symbols CharRange CharRanges

  context-free syntax

    lit ( Literal )                        -> Symbol
    cf ( Symbol )                          -> Symbol
    lex ( Symbol )                         -> Symbol
    opt ( Symbol )                         -> Symbol
    alt ( Symbol, Symbol )                 -> Symbol
    pair ( Symbol, Symbol )                -> Symbol
    sort ( Literal )                       -> Symbol
    iter ( Symbol )                        -> Symbol
    iter-star ( Symbol )                   -> Symbol
    iter-sep ( Symbol, Symbol )            -> Symbol
    iter-star-sep ( Symbol, Symbol )       -> Symbol
    iter-n ( Symbol, NatCon )              -> Symbol
    iter-sep-n ( Symbol, Symbol, Integer ) -> Symbol
    perm ( Symbols )                       -> Symbol
    set ( Symbol )                         -> Symbol
    func ( Symbols, Symbol )               -> Symbol
    varsym ( Symbol )                      -> Symbol
    layout                                 -> Symbol
    char-class ( CharRanges )              -> Symbol

  context-free syntax
    "[" {Symbol ","}* "]"                  -> Symbols
    "[" {CharRange ","}* "]"               -> CharRanges
    Integer                                -> CharRange
    range ( Integer, Integer )             -> CharRange

module Tree
imports Literals
```

```
exports
  sorts Tree Args Production

  context-free syntax
    appl ( Production, Args )            -> Tree
    Integer                              -> Tree
    lit ( Literal )                      -> Tree
    amb ( Args )                         -> Tree

    "[" {Tree ","}* "]"                  -> Args
    prod ( Symbols, Symbol, Attributes ) -> Production
    list ( Symbol )                      -> Production

module Attributes
imports Literals ATerms

exports
  sorts Attributes Attrs Attr Associativity

  context-free syntax
    no-attrs              -> Attributes
    attrs ( Attrs )       -> Attributes
    "[" {Attr ","}+ "]"   -> Attrs
    id ( module-Literal ) -> Attr
    "bracket"             -> Attr
    "reject"              -> Attr
    "prefer"              -> Attr
    "avoid"               -> Attr
    "left"                -> Associativity
    "right"               -> Associativity
    "assoc"               -> Associativity
    "non-assoc"           -> Associativity

    "assoc" Associativity -> Attr
    ATerm                 -> Attr
```

C Example generated dictionary file

An abbreviated version of the generated dictionary file for the parse tree syntax (AsFix) of the boolean `and`. Multiple similar lines have been condensed (...).

```
#include "test_dict.h"

AFun afun0;
```

```
AFun afun1;
...

ATerm patternBoolAnd = NULL;

/*
 * afun0 = appl(x,x)
 * afun1 = prod(x,x,x)
 * afun2 = cf(x)
 * afun3 = sort(x)
 * afun4 = "Bool"
 * afun5 = opt(x)
 * afun6 = layout
 * afun7 = lit(x)
 * afun8 = "and"
 * afun9 = attrs(x)
 * afun10 = assoc(x)
 * afun11 = left
 *
 * patternBoolAnd = appl(prod([cf(sort("Bool")),cf(opt(layout)),lit("and"),
 *                        cf(opt(layout)),cf(sort("Bool"))],cf(sort("Bool")),
 *                        attrs([assoc(left)])),
 *                        [<term>,<term>,lit("and"),<term>,<term>])
 *
 */

static ATermList _test_dict = NULL;

#define _test_dict_LEN 239

static char _test_dict_baf[_test_dict_LEN] = {
0x00,0x8B,0xAF,0x83,0x00,0x11,0x33,0x03,0x3C,0x5F,0x3E,0x01,0x00,0x01,0x01,0x03,
0x05,0x5B,0x5F,0x2C,0x5F,0x5D,0x02,0x00,0x1A,0x0E,0x01,0x00,0x05,0x06,0x07,0x08,
0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x02,0x01,0x02,0x02,0x5B,0x5D,0x00,0x00,
...
};



void init_test_dict()
{
  ATermList afuns, terms;

  _test_dict = (ATermList)ATreadFromBinaryString(_test_dict_baf, _test_dict_LEN);

  ATprotect((ATerm *)&_test_dict);
```

```
  afuns = (ATermList)ATelementAt(_test_dict, 0);

  afun0 = ATgetAFun((ATermAppl)ATgetFirst(afuns));
  afuns = ATgetNext(afuns);
  afun1 = ATgetAFun((ATermAppl)ATgetFirst(afuns));
...

  terms = (ATermList)ATelementAt(_test_dict, 1);

  patternBoolAnd = ATgetFirst(terms);
  terms = ATgetNext(terms);
}
```