ATerm Library User Manual

Hayco de Jong Pieter Olivier

University of Amsterdam,
Programming Research Group,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

E-mail: `jong@wins.uva.nl - olivierp@wins.uva.nl`

Abstract

This report is a manual describing the functionality provided by the ATerm Library, a high performance implementation of the ATerm datatype using maximal sharing and automatic garbage collection.

Contents

1 Introduction

This manual describes the functionality provided by the ATerm Library. The library is a high performance implementation of the ATerm datatype described in [BJO98]. If your application satisfies the following list of characteristics, the ATerm Library can probably prove to be a valuable component.

- Maximal sharing of terms is required;
- Automatic garbage collection is required;
- There is no need for destructive update of terms (e.g. the ATerm Library is not suited to implement stacks).

Typical applications that benefit from using the ATerm Library are those that manipulate (abstract syntax) trees or terms, and those that are related to the new ASF+SDF Meta-Environment [BKMO97] or the new ASF+SDF compiler [BOHK98].

This manual is intended to be a comprehensive text on how to use the ATerm Library. Readers who are interested in the design details of the ATerm Library, can find these in [BJO98].

Section 2 explains the basic steps that must be taken to initialise and work with the ATerm Library.

The level one interface described in Section 3 offers a set of operations on ATerms that is sufficiently powerful for most users, yet is concise enough to be mastered in a relatively short period of time.

Section 4 describes the level two interface which is intented for experienced ATerm users, providing a more extensive set of datastructures and functions.

The ATerm Library comes with a number of utility programs. A description of these programs can be found in Section 5.

2 Using the ATerm Library

This section explains the basics of using the ATerm Library. One design decision, basing the ATerm Library on maximally shared ATerms, has some consequences that readers should be familiar with before using the ATerm Library. Section 2.1 therefore addresses the notion of *maximal sharing* and its consequences on the use of ATerms.

The ATerm Library has a built-in garbage collector. Section 2.2 explains what this garbage collector does and when it is invoked.

To use the ATerm Library, it must first be initialised. This ensures that the garbage collector is activated and that all necessary internal structures are set up. Section 2.3 demonstrates initialisation of the ATerm Library and shows all possible commandline options. These alter the behaviour of the ATerm Library.

The garbage collector detects references to ATerms that are on the stack and in any of the registers. As it is also possible to have global references to ATerms which the garbage collector cannot detect, it is the user's responsibility to protect these references from premature erasure. Section 2.4 explains how to protect global ATerm variables and arrays of ATerms.

The ATerm Library is capable of storing ATerms in a compact, portable binary notation. Section 2.6 describes this binary ATerm format.

2.1 Maximal sharing

Although explaining the design details of the ATerm Library is beyond the scope of this manual, it is important that users realise that the ATerm Library is implemented using maximally shared ATerms. This means that before a new ATerm is created, a lookup is done to see if that term already exists. If so, that term is reused and no new term is created, resulting in maximally shared terms. This has the advantage that equality of terms can be expressed as being physically the same term, i.e., both terms must have the same memory address. It also means that, as terms can be shared without the user knowing it, they cannot be modified without creating unwanted side effects. Thus, users should not tamper with ATerms. Rather, they should use the functionality provided in the level one and level two interfaces.

2.2 Garbage collector

The ATerm Library uses garbage collection to find and delete unused ATerms. This relieves users of the burden of explicitly allocating and freeing every ATerm used in an application. Each time a new ATerm is allocated the garbage collector decides whether time has come to free resources claimed by unused ATerms or whether there still is enough space to create the requested term. Details on the design and implementation of this garbage collection can be found in [BJO98].

2.3 Initialising the ATerm Library

To initialise the ATerm Library, a call to `ATinit` must be made from `main`, passing the commandline arguments stored in `argc` and `argv`. The address of a locally created ATerm variable must also be passed. This address is used by the garbage collector to determine the bottom of the stack to be inspected for ATerms. The following code demonstrates a typical initialisation of the ATerm Library.

```
#include <stdio.h>
#include <aterm1.h>

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;                  /* Used in initialisation of library    */
    ATinit(argc, argv, &bottomOfStack); /* Initialise the ATerm library.       */
    foo();                                /* Call to code that works with ATerms. */
    return 0;                             /* End of program. */
}
```

The following commandline options can be passed to the ATerm Library:

`-at-symboltable <nr_symbols>`	Initial size of symboltable
`-at-termtable <table_class>`	Start with termtable of $2^{\text{table\_class}}$ entries
`-at-hashinfo`	write hashtable statistics to `hashing.stats` after execution
`-at-print-gc-time`	print timing information about garbage collector to `stderr` after execution
`-at-print-gc-info`	print verbose information about garbage collector to `stderr` after execution
`-at-silent`	Do not print status and version information

2.4 Protecting global ATerms and arrays of ATerms

During a garbage collect, the ATerm Library searches the stack and registers for ATerms. Therefore it finds all ATerms which were created locally in a function (*automatic* variables). All ATerm variables which were *not* created locally, must be protected by a call to `ATprotect`. For entire arrays of ATerms, `ATprotectArray` should be used. Note that not the ATerm variable itself, but its address is passed to `ATprotect`. Users of the level two interface who create a function application by building the function symbol (AFun) can also keep global references to these symbols. Such references should be protected through a call to `ATprotectAFun`.

The following code demonstrates the protection of a global ATerm and a global array of ATerms.

```
#include <stdio.h>
#include <aterm1.h>

static ATerm global_aterm;                /* global ATerm */

#define NR_ENTRIES 42                     /* arbitrary number for this example. */
static ATerm global_arr[NR_ENTRIES];      /* global ATerm array. */

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;                   /* Used in initialisation of library    */

    ATinit(argc, argv, &bottomOfStack);    /* Initialise the ATerm library.        */
    ATprotect(&global_aterm);              /* Protect the global aterm variable.   */
    ATprotectArray(global_arr, NR_ENTRIES); /* Protect the global aterm array.     */
    foo();                                 /* Call to code that works with ATerms. */

    /* End of program. */
    return 0;
}
```

2.5 Unprotecting protected ATerms

When the lifetime of an ATerm that was protected as described in Section 2.4 has expired, the ATerm should be unprotected to free any internal resources it holds. Note that these resources are not released immediately. Instead, they are no longer marked as protected to the garbage collector and as such may be freed at the next garbage collection. For each `ATprotect` function, there is a matching `ATunprotect` function.

2.6 Binary ATerm Format (BAF)

In addition to being able to parse terms in textual format and write textual representations of ATerms, the ATerm Library is also equipped to store and restore ATerms in a compact, portable binary representation. This representation is called BAF which stands for "Binary ATerm Format".

This format can be used to write a binary version of an ATerm to file, which can later be restored in a much more efficient way than would be possible had the ATerm's textual counterpart been used. This is due to the fact that textual representations have to be (re-)parsed each time they are read from file, whereas BAF directly describes how to rebuild the internal representation of an ATerm, thus skipping the parsing phase. Moreover, the maximal sharing of ATerms is exploited when writing BAF-representations, making them take up much less space than their textual representations would have needed.

Users of the ATerm Library are encouraged to use BAF representations when saving ATerms to file. BAF was designed to be platform independent, which facilitates the exchange of ATerms. The ATerm Library comes with a utility that is able to convert an ATerm's textual representation into its BAF counterpart and vice versa (see Section 5.1). This conversion makes it possible to always work with BAF representations, while still being able to look at the textual representation any time an error is suspected. It also allows conversion of textual ATerms written by programs unable to write BAF which is especially convenient when these ATerms are bulky.

Although the ATerm Library does not put any constraints on the names of ATerm-files, users are encouraged to use the extension `.baf` for BAF files. This will avoid confusion between textual representations and binary ones. Textual representations could use the extension `.trm`.

3 Level One Interface

This section explains in detail the types and functions that are defined in the level one interface. These functions are declared in `aterm1.h`. Section 3.1 reveals the types of ATerms that are used in the ATerm Library, as well as the extension to the standard `C`-types introduced in the level one interface. To avoid confusion between BAF and the ATerm type *blob*, Section 3.2 is dedicated to explaining the difference between these two notions. Finally, Section 3.3 describes all the functions that are available in the level one interface.

3.1 Level One Types

The following `C`-defines are used to represent the different ATerm types:

- `AT_FREE`: An ATerm that is marked *free* will be reused when needed. This is not a type users will want to create themselves, but it can be used to detect an ATerm that has been freed by the garbage collector.
- `AT_APPL`: An ATerm of type: function application;
- `AT_INT`: An ATerm of type: integer;
- `AT_REAL`: An ATerm of type: real;
- `AT_LIST`: An ATerm of type: list;
- `AT_PLACEHOLDER`: An ATerm of type: placeholder;
- `AT_BLOB`: An ATerm of type: binary large object;

The following `C`-types are defined in the level one interface:

- `ATbool`: a boolean value, either `ATtrue` or `ATfalse`;
- `ATerm`: an annotated term.

3.2 A note on 'blobs' and BAF

Please note that although the word *binary* is used in the abbreviations of both "blob" and BAF, these are two very different notions. A *blob* represents an ATerm that holds binary data, with no specific meaning to the ATerm Library. This notion can be used as a means of escape in case you find that you need a type of ATerm that is not on the list above. The notion of BAF is explained in Section 2.6 and refers to a specific format used for reading and writing ATerms. Thus an ATerm of type `AT_BLOB` can be saved in BAF. It could also be written in its textual representation, although this does not guarantee that the blob will be readable, after all it represents binary data.

3.3 Level One Functionality

In this section, all functions and macros (`#define`'s in `C`) are listed. Although all macros could have been implemented as a function as well, the macros listed here were chosen to be macros instead of functions for efficiency reasons. To obtain access to the level one interface, your application should `#include <aterm1.h>`.

function: `ATmake`

Summary: Create an ATerm from a string pattern and a variable number of arguments
Declaration: `ATerm ATmake(const char *pattern, ...);`
Description: Creates an ATerm given a pattern and corresponding values. The following table shows which patterns can be used, and which type of arguments should be passed if such a pattern is used.

Type	Pattern	Argument
Application	`<appl>`	`char *pattern`, arguments
Blob	`<blob>`	`int length`, `void *data`
Integer	`<int>`	`int value`
List	`<list>`	`ATerm`
Placeholder	`<placeholder>`	`char *type`
Real	`<real>`	`double value`
String	`<str>`	`char *pattern`, arguments
Term	`<term>`	`ATerm`

Types `<appl>` and `<str>` should contain a pattern consisting of the function symbol to be used and the types of the arguments. This pattern must be followed by exactly the number of arguments that are used in the pattern. The types of the arguments must match the respective types used in the pattern. Both `<appl>` and `<str>` create function applications. The difference is that `<appl>` creates one with an *unquoted* function symbol, whereas `<str>` yields a *quoted* version.

Here are some examples of `ATmake`:

```
#include <aterm2.h>

int    ival = 42;
char  *sval = "example";
char  *blob = "12345678";
double rval = 3.14;
char  *func = "f";

void foo()
{
    ATerm term[4];
    ATerm list[3];
    ATerm appl[3];

    term[0] = ATmake("<int>" , ival);       /* integer value: 42                 */
    term[1] = ATmake("<str>" , func);       /* quoted application: "f", no args */
    term[2] = ATmake("<real>", rval);       /* real value: 3.14                  */
    term[3] = ATmake("<blob>", 8, blob);    /* blob of size 8, data: 12345678    */

    list[0] = ATmake("[]");
    list[1] = ATmake("[1,<int>,<real>]", ival, rval);
    list[2] = ATmake("[<int>,<list>]", ival+1, list[1]);

    appl[0] = ATmake("<appl>", func);
    appl[1] = ATmake("<appl(<int>)>", func, ival);
    appl[2] = ATmake("<appl(<int>, <term>, <list>)>", func, 42, term[3], list[2]);

    ATprintf("appl[2] = %t\n", appl[2]);
}

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;

    ATinit(argc, argv, &bottomOfStack);
    foo();
    return 0;
}
```

function: `ATvmake`

Summary: Create an ATerm from a string pattern and a list of arguments
Declaration: `ATerm ATvmake(const char *pattern, va_list args);`
Description: See `ATmake`.

function: `ATmakeTerm`

Summary: Create an ATerm from an ATerm pattern and a variable number of arguments
Declaration: `ATerm ATmakeTerm(ATerm pat, ...);`
Description: See `ATmake`.

function: `ATvmakeTerm`

Summary: Create an ATerm from an ATerm pattern and a list of arguments
Declaration: `ATerm ATvmakeTerm(ATerm pat, va_list args);`
Description: See `ATmake`.

function: `ATmatch`

Summary: Match an ATerm against a pattern
Declaration: `ATbool ATmatch(ATerm t, const char *pattern, ...);`
Description: Matches an ATerm against a pattern, attempting to fill the 'holes'. If the ATerm matches the pattern, `ATtrue` is returned and the variables will be filled according to the pattern, otherwise `ATfalse` is returned.

The `<list>` pattern can be used to match the tail of a list as well as a variable number of arguments in a function application. Thus the first few arguments may be matched explicitly while the tail of the arguments is directed to a list.

Here are a few examples of `ATmatch`:

```
#include <aterm2.h>

void foo()
{
    ATbool result;
    ATerm  list;
    double rval;
    int    ival;

    /* Sets result to ATtrue and ival to 16. */
    result = ATmatch(ATmake("f(16)"), "f(<int>)", &ival);

    /* Sets result to ATtrue and rval to 3.14. */
    result = ATmatch(ATmake("3.14"), "<real>", &rval);

    /* Sets result to ATfalse because f(g) != g(f) */
    result = ATmatch(ATmake("f(g)"), "g(f)");

    /* fills ival with 1 and list with [2,3] */
    result = ATmatch(ATmake("[1,2,3]"), "[<int>,<list>]", &ival, &list);
}

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;

    ATinit(argc, argv, &bottomOfStack);
    foo();
    return 0;
}
```

function: `ATreadFromString`

Summary: Read an ATerm from string
Declaration: `ATerm ATreadFromString(const char *string);`
Description: This function parses a character string into an ATerm. A convenience macro `ATparse` is included in `aterm1.h`.

function: `ATreadFromTextFile`

Summary: Read an ATerm from text file
Declaration: `ATerm ATreadFromTextFile(FILE *file);`
Description: This function reads a text file and parses the contents into an ATerm.

function: `ATreadFromBinaryFile`

Summary: Read an ATerm from binary file (BAF)
Declaration: `ATerm ATreadFromBinaryFile(FILE *file);`
Description: This function reads a binary file and builds an ATerm.

function: `ATreadFromFile`

Summary: Read an ATerm from binary or text file
Declaration: `ATerm ATreadFromFile(FILE *file);`
Description: This function reads an ATerm from a file. A test is performed to see if the file is in BAF. If so, the binary term will be read, otherwise the textual representation will be parsed.

macro: `ATparse`

Summary: A convenience macro for `ATreadFromString`
Declaration: `ATerm ATparse(const char *str)`
Description: This macro is simply a shortcut to `ATreadFromString(str)`.

macro: `ATgetType`

Summary: Return the type of `term`
Declaration: `int ATgetType(ATerm term)`
Description: A macro that returns the type of an ATerm. Result is one of `AT_APPL`, `AT_INT`, `AT_REAL`, `AT_LIST`, `AT_PLACEHOLDER`, or `AT_BLOB`.

macro: `ATisEqual`

Summary: A macro that tests equality of ATerms `t1` and `t2`
Declaration: `int ATisEqual(ATerm t1, ATerm t2)`
Description: As ATerms are created using *maximal sharing* (see Section 2.1), testing equality is performed in constant time by comparing the addresses of `t1` and `t2`. Note however that `ATisEqual` only returns `ATtrue` when `t1` and `t2` are completely equal, inclusive any annotations they might have!

function: `ATwriteToTextFile`

Summary: Writes term `t` to file `f` in textual format
Declaration: `ATbool ATwriteToTextFile(ATerm t, FILE *f);`
Description: This function writes ATerm `t` to the file `f` in textual format. This term can later be read again by `ATreadFromTextFile`.

function: `ATwriteToBinaryFile`

Summary: Writes term `t` to file `f` in Binary ATerm Format (BAF)
Declaration: `ATbool ATwriteToBinaryFile(ATerm t, FILE *F);`

Description: This function writes ATerm `t` to the file `f` in BAF. This term can later be read again by `ATreadFromBinaryFile`.

function: `ATwriteToString`

Summary: Writes term `t` to a string
Declaration: `char * ATwriteToString(ATerm t);`
Description: Writes term `t` to an internal string buffer. The start of this buffer is returned. Note that the contents of this buffer are volatile and may be overwritten by any call to the ATerm Library.

function: `ATsetAnnotation`

Summary: Annotate a term with a labeled annotation
Declaration: `ATerm ATsetAnnotation(ATerm t, ATerm label, ATerm anno);`
Description: Creates a version of `t` that is annotated with annotation `anno` which is labeled by `label`.

function: `ATgetAnnotation`

Summary: Retrieves annotation of `t` with label `label`
Declaration: `ATerm ATgetAnnotation(ATerm t, ATerm label);`
Description: This function can be used to retrieve a specific annotation of a term. If `t` has no annotations, or no annotation labeled with `label` exists, `NULL` is returned. Otherwise the annotation is returned.

function: `ATremoveAnnotation`

Summary: Remove a specific annotation from a term
Declaration: `ATerm ATremoveAnnotation(ATerm t, ATerm label);`
Description: This function returns a version of `t` which has its annotation with label `label` removed. If `t` has no annotations, or no annotation labeled with `label` exists, `t` itself is returned.

function: `ATinit`

Summary: Initialise the ATerm Library.
Declaration: `void ATinit(int argc, char *argv[], ATerm *bottomOfStack);`
Description: See Section 2.3.

function: `ATprintf`

Summary: ATerm version of `printf`
Declaration: `int ATprintf(const char *format, ...);`
Description: See ATvfprintf.

function: `ATfprintf`

Summary: ATerm version of `fprintf`
Declaration: `int ATfprintf(FILE *stream, const char *format, ...);`
Description: See ATvfprintf.

function: `ATvfprintf`

Summary: ATerm version of `vfprintf`
Declaration: `int ATvfprintf(FILE *stream, const char *format, va_list args);`
Description: The functions `ATprintf`, `ATfprintf` and `ATvfprintf` are used for formatted output to file. The conversion specifiers `c, d, i, o, u, x, X, e, E, f, g, G, p, s` behave as can be expected from `(f)printf`. In addition the conversion specifiers `l, n, t` and `y` are supported:

Conversion specifier	Action
`l`	print an ATerm-list
`n`	print information about an ATerm node
`t`	print an ATerm
`y`	print the symbol of an ATerm-application

function: `ATsetErrorHandler`

Summary: Specify an errorhandler for the ATerm Library.
Declaration: `void ATsetErrorHandler((void (*handler)(const char *, va_list)));`
Description: This function can be used to set up an errorhandler for the ATerm Library. This handler will be called when an error message is sent to `ATerror`. If no handler is set, the message is printed to `stderr` through `ATfprintf` and the ATerm Library then calls `abort()`.

function: `ATerror`

Summary: Dump an errormessage and abort the ATerm Library
Declaration: `void ATerror((const char *format, ...));`
Description: If an errorhandler has been installed through a call to `ATsetErrorHandler`, this handler will be called. Otherwise `ATerror` prints a formatted message to `stderr`, through `ATvfprintf` (q.v.) followed by a call to `abort`.

function: `ATprotect`

Summary: Protect an ATerm
Declaration: `void ATprotect(ATerm *atp);`
Description: Protects an ATerm from being freed at garbage collection. See Section 2.4.

function: `ATunprotect`

Summary: Unprotect an ATerm
Declaration: `void ATunprotect(ATerm *atp);`
Description: Releases protection of an ATerm which has previously been protected through a call to `ATprotect`. See Section 2.5.

function: `ATprotectArray`

Summary: Protect an array of ATerms
Declaration: `void ATprotectArray(ATerm *start, int size));`
Description: Protects an entire array of `size` ATerms starting at `start`.

function: `ATunprotectArray`

Summary: Unprotect an array of ATerms
Declaration: `void ATunprotectArray(ATerm *start));`
Description: Releases protection of the array of ATerms which starts at `start`.

4 Level Two Interface

This section explains in detail the types and functions that are defined in the level two interface. These functions are declared in `aterm2.h`

4.1 Level Two Types

In addition to the C-types explained in Section 3.1, the level two interface also uses the following types:

- `ATermInt`: an integer value;

- `ATermReal`: a real value;
- `ATermAppl`: a function application;
- `ATermList`: a list;
- `ATermPlaceholder`: a placeholder;
- `ATermBlob`: a Binary Large OBject;
- `ATermTable`: a hashtable of ATerms;

4.2 Level Two Functionality

This section describes all functions and macros that are available in the level two interface. To obtain access to this functionality you need to `#include <aterm2.h>` instead of `<aterm1.h>` in your application.

In Section 3.3 we explained that the macros used in the level one interface could just as well have been implemented as functions, but were chosen to be implemented as macros for efficiency reasons. The same goes for all macros described in this section.

4.2.1 ATermInt

The type `ATermInt` is the ATerm representation of an integer. It abides by the rules of the `C`-type: `int`.

function: `ATmakeInt`

Summary: Build an ATermInt from an integer (`int`)
Declaration: `ATermInt ATmakeInt(int value);`

macro: `ATgetInt`

Summary: Get the integer value from an ATermInt.
Declaration: `int ATgetInt(ATermInt t)`

4.2.2 ATermReal

The type `ATermReal` is the ATerm representation of a real. It abides by the rules of the `C`-type: `double`.

function: `ATmakeReal`

Summary: Build an ATermReal from a real (`double`).
Declaration: `ATermReal ATmakeReal(double value);`

macro: `ATgetReal`

Summary: Macro to get the real value from an ATermReal.
Declaration: `double ATgetReal(ATerm t)`

4.2.3 ATermAppl

The type `ATermAppl` denotes a function application. In order to build a function application, first its function symbol (AFun) must be built. This symbol holds the name of the function application, its arity (how many arguments the function has) and whether the function name is quoted. Below are some examples of function applications and the symbols needed to create them.

```
true              - a zero arity, unquoted function application
                    sym = ATmakeAFun("true", 0, ATfalse);

"true"            - the same function application, but quoted
                    sym = ATmakeAFun("true", 0, ATtrue);

f(0)              - an unquoted function application of arity 1
                    sym = ATmakeAFun("f", 1, ATfalse);

"prod"(2, b, [])  - a quoted function application of arity 3
                    sym = ATmakeAFun("prod", 3, ATtrue);
```

function: `ATmakeAFun`

Summary: Create a function symbol (AFun)
Declaration: `AFun ATmakeAFun(char *name, int arity, ATbool quoted);`
Description: Creates an AFun, representing a function symbol with name `name` and arity `arity`. Quotedness is passed through the `quoted` argument.

function: `ATprotectAFun`

Summary: Protect a function symbol
Declaration: `void ATprotectAFun(AFun sym);`
Description: Just as ATerms which are not on the stack or in registers must be protected through a call to `ATprotect`, so must AFuns be protected by calling `ATprotectAFun`.

function: `ATunprotectAFun`

Summary: Release an AFun's protection.
Declaration: `void ATunprotectAFun(AFun sym);`

macro: `ATgetName`

Summary: Return the name of an AFun
Declaration: `char * ATgetName(AFun sym)` .

macro: `ATgetArity`

Summary: Return the arity (number of arguments) of a function symbol (AFun)
Declaration: `int ATgetArity(AFunsym)`

macro: `ATisQuoted`

Summary: Determine if a function symbol (AFun) is quoted or not
Declaration: `ATbool ATisQuoted(AFun sym)`

function: `ATmakeAppl`

Summary: Build an application from an AFun and a variable number of arguments.
Declaration: `ATermAppl ATmakeAppl(AFun sym, ...);`
Description: The arity is taken from the first argument `sym`, the other arguments to `ATmakeAppl` should be the arguments for the application. For arity $0 \ldots 6$ the corresponding `ATmakeAppl<N>` can be used instead for greater efficiency.

function: `ATmakeAppl0`

Summary: Make a function application with zero arguments
Declaration: `ATermAppl ATmakeAppl0(AFun s);`

function: `ATmakeAppl1`

Summary: Make a function application with one argument
Declaration: `ATermAppl ATmakeAppl1(AFun s, ATerm a0);`

function: `ATmakeAppl2`

Summary: Make a function application with two arguments
Declaration: `ATermAppl ATmakeAppl2(AFun s, ATerm a0, a1);`

function: `ATmakeAppl3`

Summary: Make a function application with three arguments
Declaration: `ATermAppl ATmakeAppl3(AFun s, ATerm a0, a1, a2);`

function: `ATmakeAppl4`

Summary: Make a function application with four arguments
Declaration: `ATermAppl ATmakeAppl4(AFun s, ATerm a0, a1, a2, a3);`

function: `ATmakeAppl5`

Summary: Make a function application with five arguments
Declaration: `ATermAppl ATmakeAppl5(AFun s, ATerm a0, a1, a2, a3, a4);`

function: `ATmakeAppl6`

Summary: Make a function application with six arguments
Declaration: `ATermAppl ATmakeAppl6(AFun s, ATerm a0, a1, a2, a3, a4, a5);`
Description: These functions build an application of arity zero through six, i.e. an application with $0 \ldots 6$ arguments. Use these functions to build `ATermAppl`s with small arity in favour of the generic `ATmakeAppl` described above.

macro: `ATgetAFun`

Summary: Get the function symbol (AFun) of an application
Declaration: `AFun ATgetAFun(ATermAppl appl)`

macro: `ATgetArgument`

Summary: Get the `nr`-th argument of an application
Declaration: `ATerm ATgetArgument(ATermAppl appl, int nr)`

function: `ATsetArgument`

Summary: Set the `nr`-th argument of an application to `arg`
Declaration: `ATermAppl ATsetArgument(ATermAppl appl, ATerm arg, int n);`
Description: This function returns a copy of `appl` with argument `n` replaced by `arg`.

function: `ATgetArguments`

Summary: Get a list of arguments of an application
Declaration: `ATermList ATgetArguments(ATermAppl appl);`
Description: Return the arguments of `appl` in `ATermList` format. Note: traversing the arguments of `appl` can be done more efficiently using the `ATgetArgument` macro.

function: `ATmakeApplList`

Summary: Build an application given an AFun and a list of arguments
Declaration: `ATermAppl ATmakeApplList(AFun sym, ATermList args);`
Description: Build an application from `sym` and the argument list `args`. Note: unless the arguments are already in an `ATermList`, it is probably more efficient to use the appropriate `ATmakeAppl<N>`.

function: `ATmakeApplArray`

Summary: Build an application given an AFun and an array of arguments
Declaration: `ATermAppl ATmakeApplArray(AFun sym, ATerm args[]);`

4.2.4 ATermList

function: `ATmakeList`

Summary: Create an `ATermList` of n elements
Declaration: `ATermList ATmakeList(int n, ...);`
Description: This function can be used to build an `ATermList` of n elements. The elements should be passed as arguments $1 \ldots n$.

macro: `ATmakeList0`

Summary: Macro defined to point to the empty list `[]`
Declaration: `ATermList ATmakeList0()`

function: `ATmakeList1`

Summary: Construct a list of one element
Declaration: `ATermList ATmakeList1(ATerm el0);`

function: `ATmakeList2`

Summary: Construct a list of two elements
Declaration: `ATermList ATmakeList2(ATerm el0, el1);`

function: `ATmakeList3`

Summary: Construct a list of three elements
Declaration: `ATermList ATmakeList3(ATerm el0, el1, el2);`

function: `ATmakeList4`

Summary: Construct a list of four elements
Declaration: `ATermList ATmakeList4(ATerm el0, el1, el2, el3);`

function: `ATmakeList5`

Summary: Construct a list of five elements
Declaration: `ATermList ATmakeList5(ATerm el0, el1, el2, el3, el4);`

function: `ATmakeList6`

Summary: Construct a list of six elements
Declaration: `ATermList ATmakeList6(ATerm el0, el1, el2, el3, el4, el5);`
Description: These functions build an `ATermList` of $1 \ldots 6$ elements. Longer lists should be created using the generic `ATmakeList` function described above.

macro: `ATgetLength`

Summary: Macro to get the length of list `l`
Declaration: `int ATgetLength(ATermList l)`

macro: `ATgetFirst`

Summary: Macro to get the first element of list `l`
Declaration: `ATerm ATgetFirst(ATermList l)`

macro: `ATgetNext`

Summary: Macro to get the next part (the tail) of list `l`
Declaration: `ATermList ATgetNext(ATermList l)`

macro: `ATisEmpty`

Summary: Macro to test if list `l` is empty
Declaration: `ATbool ATisEmpty(ATermList l)`

function: `ATgetPrefix`

Summary: Return all but the last element of `list`
Declaration: `ATermList ATgetPrefix(ATermList list);`

function: `ATgetLast`

Summary: Return the last element of `list`
Declaration: `ATerm ATgetLast(ATermList list);`

function: `ATgetSlice`

Summary: Get a portion (slice) of a list
Declaration: `ATermList ATgetSlice(ATermList list, int start, int end);`
Description: Return the portion of `list` that lies between `start` and `end`. Thus `start` is included, `end` is not.

function: `ATinsert`

Summary: Return `list` with `el` inserted.
Declaration: `ATermlist ATinsert(ATermList list, ATerm el);`
Description: The behaviour of `ATinsert` is of constant complexity. That is, the behaviour of `ATinsert` does not degrade as the length of `list` increases.

function: `ATinsertAt`

Summary: Return `list` with `el` inserted at position `index`
Declaration: `ATermList ATinsertAt((ATermList list, ATerm el, int index);`

function: `ATappend`

Summary: Return `list` with `el` appended to it
Declaration: `ATermList ATappend(ATermList list, ATerm el);`
Description: Note that `ATappend` is implemented in terms of `ATinsert` by making a new list with `el` as the first element and then `ATinsert`ing all elements from `list`. As such, the complexity of `ATappend` is linear in the number of elements in `list`.

When `ATappend` is needed inside a loop that traverses a list (see `parse_list1` below), behaviour of the loop will demonstrate quadratic complexity.

```
/* Example of parse_list that demonstrates quadratic complexity */
ATermList parse_list1(ATermList list)
{
    ATerm     elem;
    ATermList result = ATempty;

    /* while list has elements */
    while (!ATisEmpty(list))
    {
        /* Get head of list */
        elem = ATgetFirst(list);

        /* If elem satisfies some predicate (not shown here) then APPEND it to result */
```

```
        if (some_predicate(elem) == ATtrue)
            result = ATappend(result, elem);

        /* Continue with tail of list */
        list = ATgetNext(list);
    }

    /* Return the result list */
    return result;
}
```

To avoid this behaviour, the inner loop could use `ATinsert` instead of `ATappend` to make the new list. This will cause the resulting list to be in reverse order. A single `ATreverse` must therefore be performed, but this can be done after the loop has terminated, bringing the behaviour down from quadratic to linear complexity, but at the cost of two `ATinsert`s per element (one for each `ATinsert` in the loop, and an implicit one for each element through the use of `ATreverse`). An example is shown here in the implementation of `parse_list2`.

```
/* Example of parse_list that demonstrates linear complexity,
 * using ATinsert instead of ATappend and reversing the list
 * outside the loop just once.  */
ATermList parse_list2(ATermList list)
{
    ATerm     elem;
    ATermList result = ATempty;

    /* while list has elements */
    while (!ATisEmpty(list))
    {
        /* Get head of list */
        elem = ATgetFirst(list);

        /* If elem satisfies some predicate (not shown here) then INSERT it to result */
        if (some_predicate(elem) == ATtrue)
            result = ATinsert(result, elem);

        /* Continue with tail of list */
        list = ATgetNext(list);
    }

    /* Return result after reversal */
    return ATreverse(result);
}
```

An even further optimisation could make use of a locally allocated buffer. While traversing the list, all elements that would normally be `ATappend`ed, are now placed in this buffer. Finally, the result is obtained by starting with an empty list and `ATinsert`ing all elements from this buffer in reverse order. As the cost of allocating and freeing a local buffer is by no means marginal, this solution should probably only be applied when the loop appends more than just a few elements. The following example shows such an implementation in `parse_list3`.

```
/* Example of parse_list that demonstrates linear complexity,
 * but which avoids using ATinsert twice, by inlining ATreverse
 * using a local buffer. */
```

```
ATermList parse_list3(ATermList list)
{
    int        pos = 0;
    ATerm      elem;
    ATerm     *buffer = NULL;
    ATermList  result = ATempty;

    /* Allocate local buffer that can hold all elements of list */
    buffer = (ATerm *) calloc(ATgetLength(list), sizeof(ATerm));
    if (buffer == NULL) abort();

    /* while list has elements */
    while (!ATisEmpty(list))
    {
        /* Get head of list */
        elem = ATgetFirst(list);

        /* If elem satisfies some predicate (not shown here)
         * then add it to buffer at next available position */
        if (some_predicate(elem) == ATtrue)
            buffer[pos++] = elem;

        /* Continue with tail of list */
        list = ATgetNext(list);
    }

    /* Now insert all elems in buffer to result */
    for(--pos; pos >= 0; pos--)
        result = ATinsert(result, buffer[pos]);

    /* Release allocated resources */
    free(buffer);

    /* Return result */
    return result;
}
```

---

function: `ATconcat`

Summary: Return the concatenation of `list1` and `list2`
Declaration: `ATermList ATconcat(ATermList list1, ATermList list2);`

function: `ATindexOf`

Summary: Return the index of an ATerm in a list
Declaration: `int ATindexOf(ATermList list, ATerm el, int start));`
Description: Return the index where `el` can be found in `list`. Start looking at position `start`. Returns `-1` if `el` is not in `list`.

function: `ATlastIndexOf`

Summary: Return the index of an ATerm in a list (reverse)
Declaration: `int ATlastIndexOf(ATermList list, ATerm el, int start));`
Description: Search backwards for `el` in `list`. Start searching at `start`. Return the index of the first occurrence of `l` encountered, or `-1` when `el` is not present before `start`.

function: `ATelementAt`

Summary: Return a specific element of a list

Declaration: `ATerm ATelementAt(ATermList list, int index));`
Description: Return the element at position `index` in `list`. Return `NULL` when `index` is not in `list`.

function: `ATremoveElement`

Summary: Return `list` with one occurrence of `el` removed
Declaration: `ATermList ATremoveElement(ATermList list, ATerm el);`

function: `ATremoveAll`

Summary: Return `list` with all occurrences of `el` removed
Declaration: `ATermList ATremoveAll(ATermList list, ATerm el);`

function: `ATremoveElementAt`

Summary: Return `list` with the element at `index` removed
Declaration: `ATermList ATremoveElementAt(ATermList list, int index);`

function: `ATreplace`

Summary: Return `list` with the element at `index` replaced by `el`
Declaration: `ATermList ATreplace(ATermList list, ATerm el, int idx);`

function: `ATreverse`

Summary: Return `list` with its elements in reversed order
Declaration: `ATermList ATreverse(ATermList list));`

function: `ATfilter`

Summary: Filter entries from a list using a predicate
Declaration: `ATermList ATfilter(ATermList list, ATbool (*predicate)(ATerm));`
Description: This function can be used to filter entries that satisfy a given predicate from a list. Each item in `list` is judged through a call to `predicate`. If `predicate` returns `ATtrue` the entry is added to a list, otherwise it is skipped. The function returns the list containing exactly those items that satisfy `predicate`.

4.2.5 ATermPlaceholder

A placeholder is a special subtype used to indicate a typed hole in an ATerm. This can be used to create a term of a specific type, even though its actual contents are not filled in.

function: `ATmakePlaceholder`

Summary: Build an ATermPlaceholder of a specific type. The type is taken from the `type` parameter. See `demo_placeholder` below.
Declaration: `ATermPlaceholder ATmakePlaceholder(ATerm type);`

macro: `ATgetPlaceholder`

Summary: Get the type of an ATermPlaceholder
Declaration: `ATerm ATgetPlaceholder(ATermPlaceholder ph)`

```
#include <assert.h>
#include <aterm2.h>

/* This example demonstrates the use of an ATermPlaceholder. It creates
 * the function application "add" defined on two integers without actually
 * using a specific integer:  add(<int>,<int>).
 */
```

```
void demo_placeholder()
{
    Symbol           sym_int, sym_add;
    ATermAppl        app_add;
    ATermPlaceholder ph_int;

    /* Construct placeholder <int> using zero-arity function symbol "int" */
    sym_int = ATmakeSymbol("int", 0, ATfalse);
    ph_int = ATmakePlaceholder((ATerm)ATmakeAppl0(sym_int));

    /* Construct add(<int>,<int>) using function symbol "add" with 2 args */
    sym_add = ATmakeSymbol("add", 2, ATfalse);
    app_add = ATmakeAppl2(sym_add, (ATerm)ph_int, (ATerm)ph_int);

    /* Equal to constructing it using the level one interface */
    assert(ATisEqual(app_add, ATparse("add(<int>,<int>)")));

    /* Prints: Placeholder <int> is of type: int */
    ATprintf("Placeholder %t is of type: %t\n", ph_int, ATgetPlaceholder(ph_int));
}

int main(int argc, char *argv[])
{
    ATerm bottomOfStack;

    ATinit(argc, argv, &bottomOfStack);
    demo_placeholder();
    return 0;
}
```

4.2.6 ATermBlob

function: `ATmakeBlob`

Summary: Build a Binary Large OBject given `size` (in bytes) and `data`
Declaration: `ATermBlob ATmakeBlob(int size, void *data);`
Description: This function can be used to create an ATerm of type blob, holding the data pointed to by `data`. No copy of this data area is made, so the user should allocate this himself.

Note: due to the internal representation of a blob, size cannot exceed 2^{24} in the current implementation. This limits the size of the data area to 16 Mb.

macro: `ATgetBlobData`

Summary: Get the data section of `blob`
Declaration: `void * ATgetBlobData(ATermBlob blob)`

macro: `ATgetBlobSize`

Summary: Get the size (in bytes) of `blob`
Declaration: `int ATgetBlobSize(ATermBlob blob)`

function: `ATregisterBlobDestructor`

Summary: Register a blob-destructor function
Declaration: `void ATregisterBlobDestructor(ATbool (*destructor)(ATermBlob));`
Description: When a blob-destructor function has been registered, it will be called whenever the garbage collector deletes an ATermBlob. The destructor function can then handle the deletion

of the data area of the blob. At most 16 blob destructor functions can be registered in the current implementation.

function: `ATunregisterBlobDestructor`

Summary: Unregister a previously registered blob-destructor function
Declaration: `void ATunregisterBlobDestructor(ATbool (*destructor)(ATermBlob));`
Description: This removes a blob-destructor that has been previously installed through a call to `ATregisterBlobDestructor`.

4.3 Dictionaries

Dictionaries are datastructures which allow looking up a certain ATerm given another ATerm. The dictionary itself is also an ATerm and as such is subject to the ATerm Library rules of garbage collection. Each dictionary consists of its own list of ATerms. For each lookup in the dictionary, the list is traversed to see if the current element's key matches the one being looked up. A lookup in a dictionary demonstrates behaviour linear in the number of elements the dictionary contains. On average fifty percent of the number of elements in the dictionary are examined before a match is found (if the element is present at all). For a more efficient ATerm-to-ATerm mapping, see Tables in section 4.4.

function: `ATdictCreate`

Summary: Create a new dictionary
Declaration: `ATerm ATdictCreate();`

function: `ATdictGet`

Summary: Get the value belonging to a given key in a dictionary.
Declaration: `ATerm ATdictGet(ATerm dict, ATerm key);`

function: `ATdictPut`

Summary: Add / update a (key, value)-pair in a dictionary
Declaration: `ATerm ATdictPut(ATerm dict, ATerm key, ATerm value);`
Description: If `key` does not already exist in the dictionary, this function adds the (`key`, `value`)-pair to the dictionary. Otherwise, it updates the value to `value`. The modified dictionary is returned.

function: `ATdictRemove`

Summary: Remove the (`key`, value)-pair from the dictionary
Declaration: `ATerm ATdictRemove(ATerm dict, ATerm key);`
Description: This function can be used to remove an entry from the dictionary. If the entry was actually in the dictionary, the modified dictionary is returned. If the entry was not in the dictionary in the first place, the (unmodified) dictionary itself is returned.

4.4 Tables

The dictionaries described in Section 4.3 are in essence nothing more than linked lists, which makes them less suitable for large ATerm-to-ATerm mappings. To this end, ATerm tables were created. These are implemented as hashtables of ATerms, with each hashbucket implemented as a dictionary.

function: `ATtableCreate`

Summary: Create an ATermTable
Declaration: `ATermTable ATtableCreate(int initial_size, int max_load_pct);`
Description: This function creates an ATermTable given an initial size and a maximum load percentage. Whenever this percentage is exceeded (which is detected when a new entry is added

through `ATtablePut`), the table is automatically expanded and all existing entries are rehashed into the new table. If you know in advance approximately how many items will be in the table, you may set it up in such a way that no resizing (and thus no rehashing) is necessary. For example, if you expect about 1000 items in the table, you can create it with its initial size set to 1333 and a maximum load percentage of 75%. You are not required to do this, it merely saves a runtime expansion and rehashing of the table which increases efficiency.

function: `ATtableDestroy`

Summary: Destroy an ATermTable
Declaration: `void ATtableDestroy(ATermTable table);`
Description: Contrary to ATermDictionaries, ATermTables are themselves *not* ATerms. This means they are *not* freed by the garbage collector when they are no longer referred to. Therefore, when the table is no longer needed, the user should release the resources allocated by the table by calling `ATtableDestroy`. All references the table has to ATerms will then also be removed, so that those may be freed by the garbage collector (if no other references to them exist of course).

function: `ATtablePut`

Summary: Add / update a (key, value)-pair in a table
Declaration: `void ATtablePut(ATermTable table, ATerm key, ATerm value);`
Description: If `key` does not already exist in the table, this function adds the (`key`, `value`)-pair to the table. Otherwise, it updates the value to `value`.

function: `ATtableGet`

Summary: Get the value belonging to a given key in a table
Declaration: `ATerm ATtableGet(ATermTable table, ATerm key);`

function: `ATtableRemove`

Summary: Remove the (`key`, value)-pair from `table`
Declaration: `void ATtableRemove(ATermTable table, ATerm key);`

function: `ATtableKeys`

Summary: Get an ATermList of all the keys in a table
Declaration: `ATermList ATtableKeys(ATermTable table);`
Description: This function can be useful if you need to iterate over all elements in a table. It returns an ATermList containing all the keys in the table. The corresponding values of each key you are interested in can then be retrieved through respective calls to `ATtableGet`.

5 Utilities

This section describes the utilities that come with the ATerm Library. These utilities are automatically built when the ATerm Library is compiled and installed.

5.1 ATerm-conversion: `baffle`

This utility can be used to convert BAF to text and vice versa. It also offers the option of interpreting a BAF-file. This will show the commands which are executed to (re-)build the ATerm. This option was created to aid debugging BAF-files and is probably of little use to the regular ATerm Library user.

```
Usage: baffle [-i <input>] [-o <output> | -c] [-v] [-rb | -rt] [-wb | -wt]

    -i <input>    - Read input from file <input>        (Default: stdin)
    -o <output>   - Write output to file <output>       (Default: stdout)
```

```
-c            - Check validity of input-term
-v            - Print version information
-h            - Display help
-ri           - Write interpretation of BAF-input
-rb, -rt      - Choose between BAF and TEXT input   (Default: -rb)
-wb, -wt      - Choose between BAF and TEXT output  (Default: -wb)
```

5.1.1 `trm2baf`

A small script called `trm2baf` is included which can be used to connect a process producing textual output and a process which expects BAF input. It just sets up `baffle` with the appropriate switches and redirects `stdin` and `stdout` accordingly.

5.1.2 `baf2trm`

A small script called `baf2trm` is included which can be used to connect a process producing BAF output and a process which expects textual input. It just sets up `baffle` with the appropriate switches and redirects `stdin` and `stdout` accordingly.

5.2 Calculating an ATerm's size: `termsize`

This utility can be used to calculate three things:

- core size: the amount of memory a given ATerm needs;
- text size: the amount of memory needed to hold a textual representation of an ATerm;
- tree depth: the maximum depth of a term.

The results are written to `stdout`. As `termsize` uses `ATreadFromFile`, it can calculate the size of both textual and BAF representations of ATerms.

Usage: `termsize <` *inputfile*.

Index

References

[BJO98] M.G.J. van den Brand, H.A. de Jong, and P. Olivier. Efficient Annotated Terms. Technical report, University of Amsterdam, 1998. In preparation.

[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.

[BOHK98] M.G.J. van den Brand, P. Olivier, J. Heering, and P. Klint. Compiling Rewrite Systems: The ASF+SDF Compiler. Technical report, CWI/University of Amsterdam, 1998. In preparation.

A The ToolBus layer

The ATerm library also provides functionality needed to implement `ToolBus` tools. This section describes this `ToolBus` layer in detail.

A.1 Generation of tool interfaces

The ATerm library provides a program that takes a standard `ToolBus` *tifs* file, which is generated by the toolbus interpreter when started with the `-gentifs` option, and generates a tool interface in C for use with the ATerm library.

The generated interface consists of two files, a C source file and a C header (include) file. In the header file a number of interface functions is declared, one for each element in the input signature of the tool. It is up to the writer of the tool to provide an implementation for these functions. The generated C file contains a handler function that analyzes incoming terms from the `ToolBus`, and delegates to the appropriate interface function.

A.2 Example

Suppose we take the following simple `ToolBus` script `"test.tb"`:

```
tool testing is { command = "./testing" }

process TEST is
let
  T : testing,
  R : str
in
  execute(testing, T?) .
  snd-do(T, f1(42)) .
  snd-eval(T, f2("hello world!")) .
  rec-value(T, result(R?)) .
  printf("result: %s\n", R)
endlet

toolbus(TEST)
```

Using the command:

```
toolbus -gentifs test.tb
```

we generate a file called `"test.tifs"`. Using the command:

```
tifstoc -tool testing test.tifs
```

we generate the header file "testing.tif.h" and the source file "testing.tif.c".

"testing.tif.h" looks like this:

```
/**
 * This file is generated by tifstoc. Do not edit!
 * Generated from tifs for tool 'testing'
 * Headerfile generated at Thu Apr 22 11:37:36 1999
 */

#ifndef _TESTING_H
#define _TESTING_H

#include <atb-tool.h>

/* Prototypes for functions called from the event handler */
void f1(int conn, int);
ATerm f2(int conn, char *);
void rec_terminate(int conn, ATerm);
extern ATerm testing_handler(int conn, ATerm term);
extern ATerm testing_checker(int conn, ATerm sigs);

#endif
```

Only the functions `f1`, `f2`, and `rec_terminate` together with a simple `main` function have to be implemented to build a fully functional `ToolBus` tool. The implementation of the functions `testing_handler` and `testing_checker` has been generated in the file "testing.tif.c", which looks like this:

```
/**
 * This file is generated by tifstoc. Do not edit!
 * Generated from tifs for tool 'testing'
 * Implementation generated at Thu Apr 22 11:37:36 1999
 */

#include "testing.tif.h"

#define NR_SIG_ENTRIES  3

static char *signature[NR_SIG_ENTRIES] = {
  "rec-terminate(<testing>,<term>)",
  "rec-eval(<testing>,f2(<str>))",
  "rec-do(<testing>,f1(<int>))",
};

/* Event handler for tool 'testing' */
ATerm testing_handler(int conn, ATerm term)
{
  ATerm in, out;
  /* We need some temporary variables during matching */
  int i0;
  char *s0;
  ATerm t0;
```

```
  if(ATmatch(term, "rec-eval(f2(<str>))", &s0)) {
    return f2(conn, s0);
  }
  if(ATmatch(term, "rec-terminate(<term>)", &t0)) {
    rec_terminate(conn, t0);
    return NULL;
  }
  if(ATmatch(term, "rec-do(f1(<int>))", &i0)) {
    f1(conn, i0);
    return NULL;
  }
  if(ATmatch(term, "rec-do(signature(<term>,<term>))", &in, &out)) {
    ATerm result = testing_checker(conn, in);
    if(!ATmatch(result, "[]"))
      ATfprintf(stderr, "warning: not in input signature:\n\t%\n\tl\n", result);
    return NULL;
  }

  ATerror("tool testing cannot handle term %t", term);
  return NULL; /* Silence the compiler */
}

/* Check the signature of the tool 'testing' */
ATerm testing_checker(int conn, ATerm siglist)
{
  return ATBcheckSignature(siglist, signature, NR_SIG_ENTRIES);
}
```

This file contains an array of signature definitions (`signature`), the handler function (`testing_handler`), and the signature checker (`testing_checker`). The only thing the handler does is differentiating between the different possible input terms coming from the `ToolBus`, and delegating the actual work to the appropriate function.

As mentioned earlier, the only thing needed to implement the actual tool, is the implementation of the three interface functions `f1`, `f2`, and `rec_terminate`, and the implementation of `main` to get things going. We will first take a look at the initialization stuff that the `main` function has to do:

```
#include "testing.tif.h"

int main(int argc, char *argv[])
{
  ATerm bottomOfStack;

  ATBinit(argc, argv, &bottomOfStack);
  if(ATBconnect(NULL, NULL, -1, testing_handler) >= 0) {
    ATBeventloop();
  } else {
    fprintf(stderr, "Could not connect to the ToolBus, giving up!\n");
    return -1;
  }

  return 0;
}
```

The variable `bottomOfStack` is needed by the ATerm library to determine where to look for the stack. `argc` and `argv` are passed unchanged to `ATBinit`, so the `ToolBus` library can look for default values for things like the `ToolBus` socket address and the `ToolBus` host name.

The call to `ATBconnect` connects to a running `ToolBus`, and requires four arguments: a character string representing the tool name, a character string representing the host name of the `ToolBus` to connect to, the port number of the `ToolBus` to connect to, and a handler function. Passing `NULL`, `NULL`, and `-1` respectively as the tool name, the host name, and the port number cause the defaults for these values to be used instead.

When all goes well, the call to `ATBeventloop` starts the main `ToolBus` eventloop and the tool will be ready to receive requests from the `ToolBus`.

Now we only need the implementation of the three interface functions:

```
void f1(int conn, int value)
{
  printf("f1 called: %d\n", value);
}

ATerm f2(int conn, char *value)
{
  return ATmake("snd-value(result(<str>))", value);
}

void rec_terminate(int conn, ATerm arg)
{
  exit(0);
}
```

Note that the `conn` argument identifies the `ToolBus` connection, making it possible to distinguish which `ToolBus` made the request when connected to more than one `ToolBus` at the same time.

A.3 Functions in the `ToolBus` layer

The `ToolBus` layer offers the following functions:

function: `ATBinit`

Summary: Initialize the `ToolBus` layer and the underlying ATerm library when needed.
Declaration: `int ATBinit(int argc, char *argv[], ATerm *stack_bottom);`
Description: The return value indicates whether or not the `ToolBus` host could be found. `0` indicates that all is well, and `-1` indicates an error, in which case `errno` is set to indicate which error.

function: `ATBconnect`

Summary: Connect to a running `ToolBus`.
Declaration: `int ATBconnect(char *toolname, char *host, int port, ATBhandler h);`
Description: When `NULL` is passed as `toolname` or `host`, or `-1` is passed as `port`, default values are taken from `argv` passed to `ATBinit`. The return value indicates whether or not the connection succeeded. `-1` means that the connection failed and a positive number or zero indicates a succesful connection attempt. In this case this number is also the file descriptor of the socket connection to the `ToolBus`.

function: `ATBdisconnect`

Summary: Disconnect a from a `ToolBus`.
Declaration: `void ATBdisconnect(int fd);`

Description: This function can be used to terminate a connection that has been established earlier using `ATBconnect`

function: `ATBeventloop`

Summary: Start the tool event loop, reading and handling terms until a connection is broken.
Declaration: `int ATBeventloop(void);`
Description: This function will not return unless something goes wrong!

function: `ATBwriteTerm`

Summary: Send a term to the `ToolBus`.
Declaration: `int ATBwriteTerm(int fd, ATerm term);`
Description: A term is send using the `ToolBus` connection indicated by `fd`. This function is typically used to generate `ToolBus` events, for instance:
`ATBwriteTerm(fd, ATparse("snd-event(some-event)"));` When something goes wrong, `-1` is returned, otherwise `0` is returned.

A.4 Control flow without the `ATBeventloop`

In some situations, the `ATBeventloop` function does not offer the right flow of control for a specific application. In this case, the following set of functions can be used to create custom control flow patterns:

function: `ATBpeekOne`

Summary: Check if there is input waiting on a `ToolBus` connection
Declaration: `ATbool ATBpeekOne(int fd);`
Description: If there is input waiting on the specified connection, `ATtrue` is returned. Otherwise, `ATfalse` is returned.

function: `ATBpeekAny`

Summary: Check if there is input waiting on any `ToolBus` connection
Declaration: `int ATBpeekAny();`
Description: If there is input waiting on one of the `ToolBus` connections, the appropriate file-descriptor is returned. Otherwise, `-1` is returned. This function provides for some 'fairness' by using a round-robin scheme in traversing the connections between calls.

function: `ATBreadTerm`

Summary: Read a term from a `ToolBus` connection.
Declaration: `ATerm ATBreadTerm(int fd);`

function: `ATBhandleOne`

Summary: Read one term from a `ToolBus` connection, and call the appropriate handler. `-1` is returned when something goes wrong.
Declaration: `int ATBhandleOne(int fd);`

function: `ATBhandleAny`

Summary: Read a single term from any `ToolBus` connection and call the appropriate handler. `-1` is returned when something goes wrong.
Declaration: `int ATBhandleAny();`

function: `ATBgetDescriptors`

Summary: Gather all `ToolBus` connection file descriptor in a single descriptor set. The return value indicates the maximum value of any descriptor in the set.
Declaration: `int ATBgetDescriptors(fd_set *set);`

Note that the standard ATBeventloop can be expressed using the following code:

```
int ATBeventloop(void)
{
  int fd;
  while(ATtrue) {
    fd = ATBhandleAny();
    if(fd < 0)
      return -1;
  }
}
```

B Java implementation

Besides the C implementaton discussed upto now, we also developed a Java implementation of the ATerm datatype. We have tried to keep the interfaces of the C implementation and the Java implementation as close together as possible. Unfortunately, constraints imposed by both languages prohibit the use of a single interface for both languages. In this section we will discuss the Java interface, and highlight the differences with the C interface where appropriate.

Most differences are introduced by the fact that Java is a much more powerful language than C. Garbage collection for instance is a built-in feature of the Java language, so no `protect` and `unprotect` functions are needed in Java.

Note that the Java version is still experimental, and therefore subject to change!

B.1 Class hierarchy

Because Java is an object oriented language, we have partitioned the Java implementation in a number of classes. The resulting class hierarchy is shown in Figure 1.

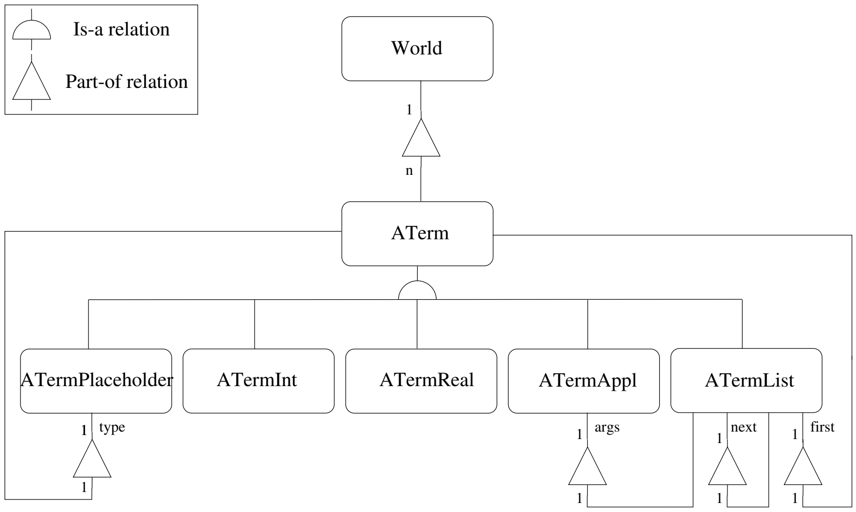

Figure 1: Simple ATerm class library

Each of the subclasses of the abstract base class `ATerm` represents one of the ATerm types. The `ATerm` class also implements the Level 1 functionality discussed earlier. This means that users who only need to use this Level 1 interface, only need to get familiar with the class `ATerm` and its methods!

The class `World` is used to implement maximal sharing, and is normally not used directly by the users of the ATerm library.

B.2 The class `ATerm`

This class implements the basic functionality to create and manipulate ATerms.

B.2.1 Creating `ATerm` objects

Because of the need to maintain maximal sharing, users of the ATerm library do not create `ATerm` objects directly using one of its constructors, but instead use one of the following static methods in the `ATerm` class:

method: `parse`

Summary: Parse an ATerm from its string representation
Declaration: `ATerm parse(String string);`
Throws: `ParseError` when `string` is not a valid term.
Description: This function takes a String as input and tries to parse it as an ATerm. If `string` represents a valid ATerm, an ATerm object is returned. If the parsing fails, a ParseError is thrown.

method: `readFromTextFile`

Summary: Read an ATerm from an `InputStream`
Declaration: `ATerm readFromTextFile(InputStream i);`
Throws: `IOException` when an I/O error occurs.
Throws: `ParseError` when the string read from tt i is not a valid term.
Description: This function takes an InputStream as input and reads one byte at a time. Note that the current implementation can only handle ASCII characters, not full Unicode.

method: `make`

Summary: Parse a pattern and fill in the placeholders in it using `arg1` through `argn`.
Declaration: `ATerm make(String pattern,Object arg1,Object arg2,...,Object argn);`
Throws: `ParseError` when `pattern` does not represent a valid term.
Description: The rules for matching are the same as in the `ATmatch` function in the C library. Note that this is not a single method but rather a set of six different methods, one for each number of `arg` arguments from zero to five. When you need to use a pattern with more than five placeholders, you must use the more general `make` method described next. If the number of placeholders in `pattern` is larger than the number of actual `arg` arguments, a `RuntimeException` is thrown. If the number of placeholders is less than the number of arguments, the excess arguments are ignored. If the type of one of the arguments does not match the placeholder type as shown in the following table, a `ClassCastException` is thrown.

Placeholder	Argument type
`<int>`	Integer
`<real>`	Double
`<str>`	String
`<appl>`	String
`<term>`	ATerm
`<list>`	ATermList

B.3 Analyzing terms

Besides creating terms, you can also analyze them using the methods described here.

method: `getType`

Summary: Retrieve the type of this ATerm object.
Declaration: `int getType();`

Description: This method returns one of the following possible constants, indicating the type of the ATerm object:

- `ATerm.INT`: The object is of type `ATermInt`
- `ATerm.REAL`: The object is of type `ATermReal`
- `ATerm.APPL`: The object is of type `ATermAppl`
- `ATerm.LIST`: The object is of type `ATermList`
- `ATerm.PLACEHOLDER`: The object is of type `ATermPlaceholder`
- `ATerm.BLOB`: The object is of type `ATermBlob`

In principle, the `instanceof` operator can be used to retrieve the same information. However, the `getType` method is much faster and the resulting integer value can be used in a `switch` statement, while the `instanceof` operator is only a boolean predicate.

method: `match`

Summary: Match a term against a String pattern, possibly yielding a list of subterms matching with placeholders.
Declaration: `Vector match(String pattern);`
Throws: `ParseError` when `pattern` does not represent a valid term.
Description: The pattern is parsed as an ATerm, and the result is matched against the object this method is called on. If the match succeeds, a `Vector` is returned containing a list of objects resulting from matches of subterms with placeholders in the pattern. If the match fails, `null` is returned.

method: `match`

Summary: Match a term against a ATerm pattern, possibly yielding a list of subterms matching with placeholders.
Declaration: `Vector match(ATerm pattern);`
Description: `pattern` is matched against the object this method is called on. If the match succeeds, a `Vector` is returned containing a list of objects resulting from matches of subterms with placeholders in the pattern. If the match fails, `null` is returned.

method: `isEqual`

Summary: Perform an equality check between two ATerms
Declaration: `boolean isEqual(ATerm term);`
Description: Because of the maximal sharing, this operation is very cheap because no deep equality check is needed.

B.3.1 Output methods

Some methods are needed to convert ATerms to `String`s and to write ATerms to `OutputStream`s and `PrintWriter`s.

method: `writeToTextFile`

Summary: Write an ATerm to an `OutputStream`.
Declaration: `void writeToTextFile(OutputStream s);`
Throws: `IOException` When an I/O error occured

method: `print`

Summary: Print an ATerm to an `OutputStream`.
Declaration: `void print(OutputStream s);`

Description: The only difference with the method `writeToTextFile` is that this method does not throw an `IOException` when something goes wrong, but instead fails silently.

method: `println`

Summary: Print an ATerm followed by a newline to an `OutputStream`
Declaration: `void println(OutputStream s);`
Description: This method does the same as the `print` method, but also prints a newline afterwards as well as flushing the `OutputStream` before returning.

method: `writeToString`

Summary: Return the `String` representation of an ATerm.
Declaration: `String writeToString();`

B.4 The class `ATermInt`

This class is used to represent integer terms. It has only one useful method:

method: `getInt`

Summary: Retrieve the integer value held by this ATerm.
Declaration: `int getInt();`

B.5 The class `ATermReal`

This class is used to represent real valued terms. It has only one useful method:

method: `getReal`

Summary: Retrieve the real value held by this ATerm.
Declaration: `double getReal();`

B.6 The class `ATermAppl`

This class is used to represent function applications. It adds the following methods to the `ATerm` base class:

method: `getName`

Summary: Retrieve the function name of this function application.
Declaration: `String getName();`

method: `isQuoted`

Summary: Ask if this function application has a quoted function name.
Declaration: `boolean isQuoted();`

method: `getArity`

Summary: Retrieve the arity of this function application.
Declaration: `int getArity();`

method: `getArguments`

Summary: Retrieve the list of function arguments of this function application.
Declaration: `ATermList getArguments();`

method: `getArgument`

Summary: Retrieve a specific function argument of this function application.
Declaration: `ATerm getArgument(int);`

B.7 The class `ATermList`

This class is used to represent lists of terms. It adds the following methods to the `ATerm` base class:

method: `isEmpty`

Summary: Check if we are dealing with the empty list.
Declaration: `boolean isEmpty();`

method: `getFirst`

Summary: Retrieve the first element of this list.
Declaration: `ATerm getFirst();`

method: `getNext`

Summary: Retrieve a list containing all elements except the first one.
Declaration: `ATermList getNext();`

method: `getLength`

Summary: Retrieve the length of this list.
Declaration: `int getLength();`

method: `indexOf`

Summary: Retrieve the index of the first occurrence of `el` after position `start`.
Declaration: `int indexOf(ATerm el, int start);`
Description: When `el` does not occur after position `start`, -1 is returned. When `start` is negative, it is interpreted as an offset from the end of the list.

method: `lastIndexOf`

Summary: Retrieve the index of the last occurrence of `el` before position `start`.
Declaration: `int lastIndexOf(ATerm el, int start);`
Description: When `el` does not occur before position `start`, -1 is returned. When `start` is negative, it is interpreted as an offset from the end of the list.

method: `concat`

Summary: Concatenate two lists.
Declaration: `ATermList concat(ATermList rhs);`

method: `append`

Summary: Append one element to a list.
Declaration: `ATermList append(ATerm el);`

method: `elementAt`

Summary: Retrieve a specific indexed element from a list.
Declaration: `ATerm elementAt(int index);`
Description: When `index` is larger than the number of elements in the list, an `IllegalArgumentException` is thrown.

method: `removeElementAt`

Summary: Return a list where the element at `index` is removed.

Declaration: `ATermList removeElementAt(int index);`
Description: When `index` is larger than the number of elements in the list, a `NullpointerException` is thrown.

B.8 The class `ATermPlaceholder`

This class is used to represent placeholder terms. It has only one useful method:

method: `getPlaceholderType`

Summary: Retrieve the type of this placeholder.
Declaration: `ATerm getPlaceholderType();`

B.9 The class `World`

This class represents a term database containing a set of terms that is maximally shared. Normally a single application only uses one instance of this class. In this case, there is no need to call any of the methods in this class, because the `ATerm` class and its subclasses provide a much cleaner interface to the functionality offered by the `World` class. In some rare cases however, an application might need to create more than one `World` instance. This could be the case for instance when different Threads need to manipulate diverse terms and the overhead of locking and unlocking the same `World` object by both threads for every term creation is too big.

A complete discussion of the `World` class falls outside the scope of this document.

C ToolBus support in Java

The Java layer also has ToolBus support situated in the package `aterm.toolbus`. This section explains how to use this support to build toolbus tools.

C.1 Generation of tool interfaces

Just as in the C case, the Java version of the ATerm library provides a program that takes a standard `ToolBus` *tifs* file, which is generated by the toolbus interpreter when started with the `-gentifs` option, and generates a a tool interface in Java for use with the ATerm library. This program is called `tifstojava`.

The generated interface consists of an abstract class that must be subclassed to implement a fully functional ToolBus tool. The only thing that has to be done in the subclass is to implement a number of abstract methods that correspond to the elements of the input signature of the tool. The generated superclass takes actually reads the input terms received from the ToolBus and delegates the actual work to the corresponding interface methods.

Besides the methods corresponding to elements from the tool input signature, two other methods are generated:

```
abstract void recTerminate(ATerm arg) throws ToolException;
abstract void recAckEvent(ATerm event) throws ToolException;
```

`recTerminate` is called when the ToolBus terminates the connection with the tool. `recAckEvent` is called when an event acknowledgement is received from the ToolBus. This method is only generated when the ToolBus script actually contains `snd-ack-event` atoms for this tool.

The `tifstojava` program can be called as follows:

```
tifstojava -tool <tool> -tifs <tifs> [-class <classname> -package <package>]
```

Normally the class name is constructed from the tool name, but this can be overridden using the `-class` option. By default no package declaration is generated, but this can be changed using the `-package` option.

C.2 Example

Suppose we take the same simple ToolBus script "test.tb" as it was used in the C version:

```
tool testing is { command = "java-adapter -class TestingTool" }

process TEST is
let
  T : testing,
  R : str
in
  execute(testing, T?) .
  snd-do(T, f1(42)) .
  snd-eval(T, f2("hello world!")) .
  rec-value(T, result(R?)) .
  printf("result: %s\n", R)
endlet

toolbus(TEST)
```

Using the command:

```
toolbus -gentifs test.tb
```

we generate a file called "test.tifs". Using the command:

```
tifstojava -tool testing -tifs test.tifs}
```

we generate the Java source file "TestingTif.java" that looks like this:

```
abstract public class TestingTif extends aterm.tool.Tool
{
  ...

  // Override these abstract methods to handle incoming ToolBus terms
  abstract void f1(int i0) throws ToolException;
  abstract ATerm f2(String s0) throws ToolException;
  abstract void recTerminate(ATerm t0) throws ToolException;

  ...
}
```

This class inherits the Runnable interface and the run method from its superclass Tool. This method contains the actual ToolBus event loop.

A fully functional ToolBus tool can now be implemented by subclassing the generated class TestingTif:

```
import java.io.IOException;
import java.net.UnknownHostException;
import aterm.tool.ToolException;

public class TestingTool extends TestingTif
{
  public TestingTool(String args) throws UnknownHostException
  {
    super(args);
  }
```

```
  void f1(int i)
  {
     System.out.println("f1 called: " + i);
  }

  ATerm f2(String s0)
  {
    return ATerm.make("snd-value(result(<str>))", s0);
  }

  void recTerminate(ATerm t0)
  {
    System.exit(0);
  }

  public static void main(String[] args)
    throws UnknownHostException, IOException, ToolException
  {
    TestingTool T = new TestingTool(args);
    T.connect();
    T.run();
  }
}
```

This tool has exactly the same functionality as the test tool in the C example. Notice how the `main` function is used to construct a new instance of `TestingTool`. This instance is then connected to the `ToolBus` and the `ToolBus` event loop is started.

C.3 The class Tool

The only really interesting class in the Java `ToolBus` support is the `Tool` class. The `Tool` class has three constructors:

method: `Tool`

Summary: Create a simple tool that will connect to a ToolBus instance on the localhost on the standard ToolBus port (8998)
Declaration: `Tool(String name);`
Throws: `UnknownHostException` when something is seriously wrong with the network support of the local machine.

method: `Tool`

Summary: Create a tool that will connect to a ToolBus instance on a host with the given internet address on the given port.
Declaration: `Tool(String name, InetAddress address, int port);`
Throws: `UnknownHostException` when the host cannot be found.

method: `Tool`

Summary: Create a tool that will connect using a configuration that is taken from a list of arguments.
Declaration: `Tool(String[] args);`
Description: The arguments should contain at least the option `-TB_TOOL_NAME <name>` and can optionally contain:

- `-TB_HOST <host>` indicating the host to connect to;

- -TB_PORT <port> indicating the port to connect to;
- -TB_TOOL_ID <id> indicating the tool id when already assigned by the ToolBus;
- -TB_VERBOSE to print debugging information to stderr.

The other public methods in this class are:

method: connect

Summary: Connect to a running ToolBus instance.
Declaration: void connect();
Throws: IOException when the connection cannot be established
Throws: ToolException when something goes wrong during the handshake with the ToolBus.

method: run

Summary: Start the ToolBus eventloop.
Declaration: void run();
Description: This method does not return until the connection with the ToolBus has been terminated.

method: send

Summary: Send a term to the ToolBus.
Declaration: void send(ATerm term);
Throws: ToolException when the write to the ToolBus fails.

method: event

Summary: Send an event to the ToolBus.
Declaration: void event(ATermAppl appl);
Throws: ToolException when the write to the ToolBus fails.
Description: This is a convenience method that simplifies the sending of events to the ToolBus.
event((ATermAppl)ATerm.parse("an-event")) is equivalent to send(ATerm.parse("snd-event(an-event)")).

method: post

Summary: Post an event to the ToolBus.
Declaration: void post(ATermAppl appl);
Throws: ToolException when the write to the ToolBus fails.
Description: This method is equivalent to event, except that events are queued when not yet acknowledged. This means that this method can be used to send events regardless of whether the previous event of this type has been acknowledged. The event will only be sent after the acknowledgement of the previous event has been processed.

method: setVerbose

Summary: Change the verbose mode of this tool.
Declaration: void setVerbose(boolean on);
Description: When in verbose mode, debug information is printed on stderr.