ATerm Library User Manual

Hayco de Jong Pieter Olivier

Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

E-mail: jong@cwi.nl - olivierp@cwi.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 [BJKO00]. 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 (the ATerm Library is not suited to implement e.g. 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 [BJKO00].

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 [BJKO00].

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`.

2.7 Shared Textual ATerm Format (TAF)

In addition to the binary aterm format there is also a textual aterm format which supports maximal sharing but uses a much less complex algorithm than the one used to encode and decode BAF files. This results in files that are somewhat larger than their baf counterparts, but are often (if the terms contain redundancy) significantly smaller than their unparsed form.

TAF files always start with a '!' character to distinguish them from aterm formats. The format uses abbreviations to refer to previously written terms. An abbreviation consists of a hash character ('#') followed by a number in encoded using the Base64 Alphabet (see RFC2045).

Each term whose unparsed representation would take up more bytes than the textual representation of the next available abbreviation is assigned such an abbreviation *after* it has been written. Subsequent occurences of this term are then written by emitting the abbreviation instead of the term itself.

For example the term `f(test,test)` is represented as `!f(test,#A)` in TAF, whereas `f(a,a)` is respresented as `!f(a,a)` because `test` is longer than its abbreviation `#A`, but `a` is not.

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: `ATreadFromBinaryString`

Summary: Read a ATerm from a string in BAF format.
Declaration: `ATerm ATreadFromBinaryString(const char *string, int size);`
Description: This function decodes a BAF character string into an ATerm.

function: `ATreadFromSharedString`

Summary: Read a ATerm from a string in TAF format.
Declaration: `ATerm ATreadFromSharedString(const char *string, int size);`
Description: This function decodes a TAF character string into an ATerm.

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: `ATreadFromSharedTextFile`

Summary: Read an ATerm from a shared text file (TAF)
Declaration: `ATerm ATreadFromSharedTextFile(FILE *file);`
Description: This function reads a shared text file and buils 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, TAF, or plain text.

function: `ATreadFromNamedFile`

Summary: Read an ATerm from named binary or text file
Declaration: `ATerm ATreadFromNamedFile(char *filename);`
Description: This function reads an ATerm file `filename`. A test is performed to see if the file is in BAF, TAF, or plain text. "-" is standard input's filename.

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: `ATwriteToSharedTextFile`

Summary: Writes term `t` to file `f` in shared Textual ATerm Format (TAF)
Declaration: `long ATwriteToSharedTextFile(ATerm t, FILE *f);`
Description: This function writes ATerm `t` to the file `f` in TAF, and returns the number of characters written. This term can later be read again by `ATreadFromSharedTextFile`.

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: `ATWriteToNamedTextFile`

Summary: Writes term `t` to file named `filename` in textual format
Declaration: `ATerm ATWriteToNamedTextFile(ATerm t, char *filename);`
Description: This function writes ATerm `t` in textual representation to file `filename`. "-" is standard output's filename.

function: `ATWriteToNamedBinaryFile`

Summary: Writes term `t` to file named `filename` in Binary ATerm Format (BAF)
Declaration: `ATerm ATWriteToNamedBinaryFile(ATerm t, char *filename);`

Description: This function writes ATerm `t` in textual representation to file `filename`. "-" is standard output's filename.

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: `ATwriteToSharedString`

Summary: Writes term `t` to a shared text string in TAF
Declaration: `char * ATwriteToSharedString(ATerm t, int *len);`
Description: Writes term `t` to an internal string buffer in TAF. The start of this buffer is returned, and the length of the resulting string is stored in len. Note that the contents of this buffer are volatile and may be overwritten by any call to the ATerm Library.

function: `ATwriteToBinaryString`

Summary: Writes term `t` to a shared text string in BAF
Declaration: `char * ATwriteToBinaryString(ATerm t, int *len);`
Description: Writes term `t` to an internal string buffer in BAF. The start of this buffer is returned, and the length of the resulting string is stored in len. 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 `a, h, l, n` and `t` are supported:

Conversion specifier	Action
`a`	print the symbol of an ATerm-application
`h`	print the MD5 checksum of an ATerm
`l`	print an ATerm-list
`n`	print information about an ATerm node
`t`	print an ATerm

function: `ATsetWarningHandler`

Summary: Specify a warninghandler for the ATerm Library
Declaration: `void ATsetWarningHandler((void (*handler)(const char *, va_list)));`
Description: Sets a warninghandler for the ATerm Library. This handler will be called when an error message is sent to `ATwarning`.

function: `ATwarning`

Summary: Issue a warning message
Declaration: `void ATwarning((const char *format, ...));`
Description: If an errorhandler has been installed through a call to `ATsetWarningHandler`, this handler will be called. Otherwise `ATwarning` uses `ATvfprintf` to print a formatted message to `stderr` and returns.

function: `ATsetErrorHandler`

Summary: Specify an errorhandler for the ATerm Library
Declaration: `void ATsetErrorHandler((void (*handler)(const char *, va_list)));`
Description: Sets an errorhandler for the ATerm Library. This handler will be called when an error message is sent to `ATerror`.

function: `ATerror`

Summary: Issue an errormessage and exit 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` uses `ATvfprintf` to print a formatted message to `stderr` and `exits` with errorcode 1.

function: `ATsetAbortHandler`

Summary: Specify an aborthandler for the ATerm Library
Declaration: `void ATsetAbortHandler((void (*handler)(const char *, va_list)));`
Description: Sets an aborthandler for the ATerm Library. This handler will be called when an error message is sent to `ATabort`.

function: `ATerror`

Summary: Issue an errormessage and abort the ATerm Library
Declaration: `void ATerror((const char *format, ...));`
Description: If an aborthandler has been installed through a call to `ATsetAbortHandler`, this handler will be called. Otherwise `ATabort` uses `ATvfprintf` to print a formatted message to `stderr` calls `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;
- `ATermIndexedSet`: a set of ATermswhere each element has a unique index.

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(AFun sym)`

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...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: `ATgetTail`

Summary: Return the sublist from `start` to the end of `list`.
Declaration: `ATermList ATgetTail(ATermList list, int start);`

function: `ATreplaceTail`

Summary: Replace the tail of `list` from position `start` with `newtail`

Declaration: `ATermList ATreplaceTail(ATermList list, ATermList newtail, int start);`

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 efficiently implemented using a hash table requiring approximately 16 bytes per stored entry, assuming that the hash table is filled for 50%.

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 using `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: `ATtableReset`

Summary: Reset an ATermTable
Declaration: `void ATtableReset(ATermTable table);`
Description: This function resets an ATermtable, without freeing the memory it occupies. Its effect is the same as the subsequent execution of a destroy and a create of a table, but as no memory is released and obtained from the C memeory management system this function is generally cheaper. but if subsequent tables differ very much in size, the use of `ATtableDestroy` and `ATtableCreate` may be prefered, because in such a way the sizes of the table adapt automatically to the requirements of the application.

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`.

4.5 Indexed sets

The data type `ATermIndexedSet` provides a mapping or table from `ATerms` to numbers, where it attempts to assign the numbers from one upwards subsequently to each entered term. If a number is assigned to a term, the term will remain assigned until the term is removed from the table. When assigning numbers to newly entered elements, numbers previously assigned to elements that have been removed are used first.

This datatype can be used for different purposes. In the first place one can make a mapping from `ATerms` to elements in any arbitrary domain D with it. By entering the `ATerms` in an `ATermIndexedSet` each `ATerm` gets a subsequent number. These numbers can be used as entries in a table to obtain the element of sort D belonging to the `ATerm`.

Another type of application is the use as a set. Suppose that a sequence of `ATerms` must be dealt with. Suppose that the sequence can contain identical `ATerms`, and that each unique `ATerm`

needs to be treated only once. Then each treated `ATerm` can be entered in the indexed set. For each to be investigated `ATerm` one inspection of the indexed set suffices to know whether this `ATerm` has already been considered. A particular instance of this kind of application is the exploration of state spaces, where each state is represented by an `ATerm`.

The implementation `ATermIndexedSet`s and `ATermTable`s are strongly connected. The implementation is quite efficient both in time and space, only requiring 12 bytes for each entry in an indexed set, if the hash table, which forms it core, is half full.

function: `ATindexedSetCreate`

Summary: Create a new `ATermIndexedSet`
Declaration: `ATermIndexedSet ATindexedSetCreate(long initial_size, int max_load_pct);`

Description: This function creates an `ATermIndexedSet` with approximately the size `initial_size`, where it guarantees that the internal hash table, will be filled up to `max_load_pct` percent. If needed, the size of the hash table is dynamically extended to hold the entries inserted into it. If extension of the hash table fails due to lack of memory. it is attempted to fill the hash table up to 100%. All elements entered into the indexed set are automatically protected. Note that for each `ATindexedSetCreate` an `ATindexedSetDestroy` must be carried out to free memory, and to allow inserted elements to be released by the automatic garbage system of the `ATerm` library. Carrying out a `ATindexedSetReset` does not free the memory, but allows inserted elements to be garbage collected.

function: `ATindexedSetDestroy`

Summary: This function releases all memory occupied by the `ATermIndexedSet`.
Declaration: `void ATindexedSetDestroy(ATermIndexedSet set);`

function: `ATindexedSetReset`

Summary: Clear the hash table in the set.
Declaration: `void ATindexedSetReset(ATermIndexedSet set);`
Description: This function clears the hash table in the set, but does not release the memory. Using `ATindexedSetReset` instead of `ATindexedSetDestroy` is preferable when indexed sets of approximately the same size are being used.

function: `ATindexedSetPut`

Summary: Enter `elem` into the set.
Declaration: `long ATindexedSetPut(ATermIndexedSet set, ATerm elem, ATbool *new);`
Description: This functions enters `elem` into the set. If `elem` was already in the set the previously assigned index of `elem` is returned, and `new` is set to false. If `elem` did not yet occur in `set` a new number is assigned, and `new` is set to true. This number can either be the number of an element that has been removed, or, if such a number is not available, the lowest non used number is assigned to `elem` and returned. The lowest number that is used is 0.

function: `ATindexedSetGetIndex`

Summary: Find the index of `elem` in `set`
Declaration: `long ATindexedSetGetIndex(ATermIndexedSet set, ATerm elem);`
Description: The index assigned to `elem` is returned, except when `elem` is not in the set, in which case the return value is a negative number.

function: `ATindexedSetGetElem`

Summary: Retrieve the element at `index` in `set`
Declaration: `ATerm ATindexedSetGetElem(ATermIndexedSet set, long index);`
Description: This function must be invoked with a valid index and it returns the `elem` assigned to this index. If it is invoked with an invalid index, effects are not predictable.

function: `ATindexedSetRemove`

Summary: Remove `elem` from `set`
Declaration: `void ATindexedSetRemove(ATermIndexedSet set, ATerm elem);`
Description: The `elem` is removed from the indexed set, and if a number was assigned to `elem`, it is freed to be reassigned to an element, that may be put into the set at some later instance.

function: `ATindexedSetElements`

Summary: Retrieve all elements in `set`
Declaration: `ATermList ATindexedSetElements(ATermIndexedSet set);`
Description: A list with all valid elements stored in the indexed set is returned. The list is ordered from element with index 0 onwards.

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 between the different ATerm formats: `text`, `baf`, and `taf`.

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

    -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
    -rb, -rt, -rs - Choose between BAF, TEXT, and TAF input (Default: autodetect)
    -wb, -wt       - Choose between BAF, TEXT and TAF output (Default: -wb)
```

Some small scripts are included which can be used to connect a process producing one ATerm format to a process which expects another. These scripts just set up `baffle` with the appropriate switches and redirect `stdin` and `stdout` accordingly. These scripts are appropriately called: `trm2baf`, `baf2trm`, `trm2taf`, `taf2trm`, `baf2taf`, and `taf2baf`.

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*.

5.3 Calculating MD5 checksum of an ATerm: `atsum`

This utility can be used to print the MD5 checksum of the TAF representation of an ATerm. The algorithm used is the RSA Data Security, Inc. MD5 Message-Digest Algorithm (see RFC1321).

Usage: `atsum [`*`inputfile`*`]`.

5.4 Calculating differences between two ATerms: `atdiff`

This utility compares two terms and prints a template term that covers the common parts containing `<diff>` placeholders for subterms that differed, and a list of their differing subterms.

```
Usage: atdiff [<options>] <file1> <file2>
Options:
    --nodiffs | --diffs <diff-file>              (default: stdout)
    --notemplate | --template <template-file>    (default: stdout)
```

Index

References

[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.

[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.

B.1 Interfaces and Implementation

The interface `ATerm` defines functionality relevant for all ATerm subtypes. Each of these ATerm subtypes has its own interface, describing the additional functionality relevant for that particular subtype.

An interface `ATermFactory` describes the various methods used to create new ATerm objects. It is used to implement maximal sharing.

A complete description of these interfaces can be found in the file `java-api.ps` which is distributed with this manual.

The ATerm library distribution so far comes with a single implementation of the ATerm interfaces. This implementation is a "pure" Java one, but plans exist to experiment with an implementation using the Java Native Interface (JNI) to build a layer of Java code on top of the C implementation.

A very basic example that shows the creation of some ATerms and how to read one from a stream is shown below.

```
import java.io.*;
import aterm.*;

public class Basic
{
  private ATermFactory factory;

  public static final void main(String[] args) throws IOException {
    Basic basic = new Basic(args);
  }

  public Basic(String[] args) throws IOException {
    factory = new aterm.pure.PureFactory();

    ATermInt i = factory.makeInt(42);
    System.out.println("i = " + i);
```

```
    AFun fun = factory.makeAFun("foo", 2, false);
    ATermAppl foo = factory.makeAppl(fun, i, i);
    System.out.println("foo = " + foo);

    ATerm t = factory.parse("this(is(a(term(0))))");
    System.out.println("t = " + t);

    try {
      ATerm input = factory.readFromFile(System.in);
      System.out.println("You typed a valid term: " + input);
    } catch (ParseError error) {
      System.out.println("Your input was not a valid term!");
    }
  }
}
```

B.2 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.

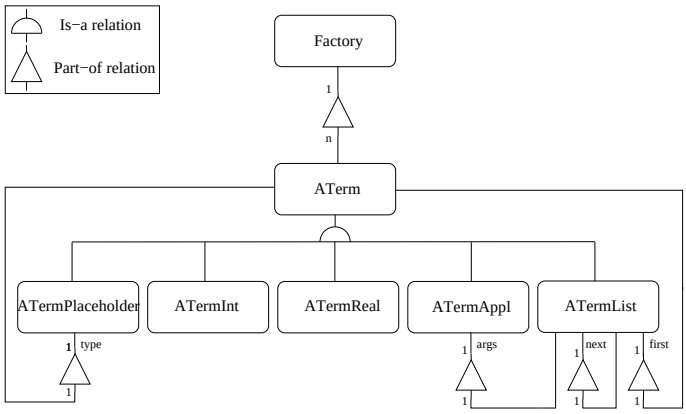

Figure 1: Simple ATerm class library

C Connecting Java tools

This section shows how to write a simple Java application and which steps need to be taken to connect it to the ToolBus.

The example tool is a simple Java Application which pops up a GUI with a button that can be pressed by the user. Further, it contains a method `msg` which can be called by the ToolBus. This method prints the string that was passed and returns a count indicating the message number.

In order to get from the ToolBus script `example.tb` to the generated `.java` files, we follow steps similar to the procedure to connect a `C` program to the ToolBus.

- Generate Tool Interface file `example.tifs`

```
~/tmp > toolbus -gentifs example.tb
Tool interfaces written to 'example.tifs'
```

- Generate Java code for tool `example`

```
~/tmp > tifstojava -tool example -tifs example.tifs
generating file ExampleTif.java
generating file ExampleTool.java
generating file ExampleBridge.java
```

As you can see, three files are generated for our Example tool:

- `ExampleTif.java` the Java interface describing our Tool's functionality.
- `ExampleTool.java` an *abstract* implementation of `ExampleTif.java` containing just enough code to check the input/output signature of our tool and handle incoming messages by dispatching them to the correct method.
- `ExampleBridge.java` a *concrete* class extending `ExampleTool.java` which can be used as a bridge between your Tool class and the ToolBus.

Remember that Java only allows extention of a single class, but *does* allow your class to implement multiple interfaces. Using the generated `Bridge` allows your class to extend another class, but still save you the arduous task of extending the generic *AbstractTool* class for each new tool. Have a look at the source code of `example.tb` and `Example.java` below.
`example.tb` looks like this:

```
tool example is { command="java-adapter -class Example" }

process EXAMPLE is
let
  T : example,
  Value : int,
  Name  : str
in
  execute(example, T?) .
  (
    snd-eval(T, msg("Hello World!")) delay(sec(3)) .
    rec-value(T, count(Value?)) .
    printf("Hello number %d\n", Value)
  ) * delta
  ||
  (
    rec-event(T, button(Name?)) .
    printf("Button pressed: %s\n", Name) .
    snd-ack-event(T, button(Name))
  ) * delta
endlet

toolbus(EXAMPLE)
```

`Example.java` looks like this:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

import aterm.*;
```

```
public class Example
  extends Frame
  implements ExampleTif, ActionListener
{
  private ATermFactory factory;
  private ExampleBridge bridge;
  private Button button;
  private int count;

  public static final void main(String[] args)
    throws IOException
  {
    Example example = new Example(args);
  }

  public Example(String[] args)
    throws IOException
  {
    factory = new aterm.pure.PureFactory();

    // Build the user interface: just a single button
    button = new Button("Button");
    button.addActionListener(this);
    add(button);
    pack();
    show();

    // Create the bridge that will forward incoming messages
    // to method calls in this Example object
    bridge = new ExampleBridge(factory, this);

    // Initialize vital parameters, like the ToolBus TCP/IP port, tool name,
    // etc. that are passed to us using the command line arguments.
    bridge.init(args);

    // Actually establish the connection with the ToolBus
    bridge.connect();

    // Start the tool event loop
    bridge.run();
  }

  public void actionPerformed(ActionEvent event)
  {
    if (event.getSource() == button) {
      // When the user presses the button, we send an event to the ToolBus
      bridge.postEvent(factory.make("button(<str>)", button.getLabel()));
    }
  }

  public ATerm msg(String message)
  {
    // Print the incoming message
    System.out.println("Example tool received msg: " + message);
```

```
    // Increase the counter and return the current value to the ToolBus
    return factory.make("snd-value(count(<int>))", new Integer(count++));
  }

  public void recAckEvent(ATerm event)
  {
    // This simple tool ignores event acknowledgements
  }

  public void recTerminate(ATerm arg)
  {
    // Just exit when the ToolBus terminates
    System.exit(0);
  }
}
```