Testing the ATerm library

Lem, P.

Award date:
2008

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Testing the ATerm library

By
Peter Lem

SUPERVISOR

Prof. Dr. M.G.J. van den Brand

Eindhoven, July 2008

Preface

This thesis concludes my Master of Science studies at the department of mathematics and computer science at the Technische Universiteit Eindhoven.

I would like to thank Judi Romijn for helping me find a suited graduation assignment.

Many thanks go out to my supervisor Mark van den Brand for his great input, his friendly guidance and support throughout the entire project. Moreover, thanks go out to Mark for his countless useful feedback during the writing of this text.

Another word of gratitude goes to Mohammad Mousavi for joining my assessment committee and giving me feedback on my work.

I would also like to thank my officemates for creating a great atmosphere to work in. Lastly, I would like to thank my friends and my family for their moral support and motivation.

–Peter Lem,
July 2008

Abstract

Testing is an important technique to measure and ensure software quality. It is part of almost any software project. The testing phase of typical projects takes up to 50% of the costs of the total project and therefore contributes significantly to the project costs.

Testing can be quite difficult without requirements or specification of the software that is being tested. Specifically this is the case for the ATerm library, which is a library handling Annotated Terms. In this thesis a framework is presented by which specifications can be created easily and testcases are generated automatically (based on the specifications). An implementation of this framework is used to test 2 smaller libraries after which the C version of the ATerm library itself is tested. A notable technique is using context dependent rule coverage on the grammar of the ATerm language to create a set of representative values.

Contents

Chapter 1

Introduction

1.1 Motivation

Engineering disciplines combine design and construction activities with activities that check intermediate and final products so that defects can be identified and removed. Software engineering is no exception to this, to ensure high quality, software verification has to be performed at various stages throughout the development process. Appropriate verification activities depend on the engineering discipline, the construction process, the final product and quality requirements [31].

One technique that can be used in the developement process is software verification through correctness proof. Software testing, or the process of executing a program or system with the intent of finding errors, is another alternative for verifying a software system. Software testing can be appropriately used in conjunction with correctness proofs, but software testing is used as the only verification technique most of the time.

Regression testing is a software testing technique which ensures that new or modified features do not cause current releases to regress after incorporating fixes into product. A simple approach to regression testing consists of reexecuting all test cases designed for previous versions. However some test cases may not be reexecutable without modifications since functions have changed. Other test cases may be obsolete since they test functionality that has been removed. A good quality test suite must be maintained across system versions.

ATerms (short for Annotated Terms) are an abstract data type designed for the exchange of tree-like data structures between distributed applications. ATerms and the ATerm library were designed and implemented in the beginning of 1998 at the CWI (Centrum voor Wiskunde en Informatica). The ATerm library uses maximal subterm sharing: only new terms are created. For this purpose a hashtable is used which has pointers to every term that is created. The C version (a Java version also exists) of the ATerm library also has an automated garbage collector. Local terms which are not used with a certain frequency are removed automatically. Global terms have to be protected in order for them not to be removed by the garbage collector.

This project is aimed at improving the overall testing process for the ATerm library. The source code of the library contains one file which contains all the tests for the library. We will evaluate these tests and provide a better test suite for the library.

1.2 Research questions

The current stable C version of the ATerm library contains 1 source file which tests some of the functions of the library. These tests are not documented. The main question for this thesis is "How can the overall testing process of the ATerm library be improved?". To structure the answer to this question, the following research questions will be addressed.

- "How good are the current set of tests?"
- "Should the test set be revised, and according to which guidelines?"

The first question can be split into several subquestions

1. "How much statement coverage does the current regression test set yield of the library code?"
2. "Which features of the library are covered with the current regression test set?"
3. "How can full coverage of the above two be obtained? Is this feasible?"

To answer these questions we executed the test file with a coverage tool to get insights into the amount of code that is covered and the parts of the library that are executed. Further code analysis answered the question if it is feasible to achieve full statement coverage. Other types of coverage were also used to determine which tests should be added to the test suite. An improved test suite was developed using an analysis of the current set of tests. Known software testing techniques were used to create appropriate test cases and input values. The resulting test suite for the ATerm library can be reused easily whenever new versions of the library are created. The test suite was created in such a way that the techniques are not specific to the ATerm library, but can be reused for other software libraries and programs as well.

1.3 Thesis outline

Chapter 2 gives an introduction to software testing and desribes terms and formalisms which are relevant to this project. In Chapter 3 explains the ATerm library and illustrate its use. The current tests were executed on the library source code so that they could be analyzed in terms of coverage of the library. Chapter 4 explains the used testing approach. The notion of universal quantification is introduced and we show how to use it not only as a specification tool for a software program, but also as a basis for generating test cases. Chapter 5 describes an implementation of the framework we described in Chapter 4. Chapter 6 discusses the first results of using UQT to test the ATerm library. In Chapter 7 the results of using UQT to test 2 C libraries, a linked list library and a data structure library, are discussed. By using the framework to test some other libraries, we show that the implementation is not too specific to the ATerm library. The results of testing these libraries lead to some modifications to the framework which we discuss in Chapter 8. In that chapter, we also introduce the notion of context dependent rule coverage and illustrate how it can be used to generate test data. Chapter 9 contains the final results of testing the ATerm library, including a list of bugs and coverage metrics. Finally, chapter 10 contains the overal results and conclusions of the project.

Chapter 2

Software testing

2.1 Introduction

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979: "Software testing is the process of executing a program or system with the intent of finding errors [29]". In 1988 it was first stated that software testing is also used to establish the quality of the software [20]. In the context of commercial enterprises the goal of software testing is to ensure that software is production ready [27]. Interesting stories about the impact and costs of inadequately tested software can be found in [14].

When we talk about testing, we usually mean execution-based testing; we investigate the results of a certain execution of a program. We can also "test" a program without executing it, for instance by performing a code walkthrough or code inspection. In this document we will talk about execution-based testing only, unless mentioned otherwise.

It is impossible to test a program for all possible input values, since the number of possible input values can become extremely large. Therefore some input values have to be selected, which we used to test the program. We gain a certain amount of confidence in the proper working of the software after the selected input values have been tested (successfully), but we cannot be sure that the program works correctly for *all* input values. We discuss how to select representative input values in section 2.3.1.

Most software is very complex. This makes the testing software a difficult job. Over time improvements have been made in software development tools. The consequence of this is that developers can create more code more easily, which usually means more bugs and more complex programs to test.

2.2 Types of testing

Testing can be divided into 2 groups, based on the way the test cases where generated:

- **Black box testing** is testing without knowledge of the internal workings of the item being tested. The tester only knows what the legal inputs and the expected outputs should be, but not how the program actually computes those outputs. Black box testing is considered as testing with respect to the specifications. The tester and the programmer can be independent of one another, avoiding programmer bias toward his own work. The research in black-box testing mainly focuses on how to maximize the effectiveness of testing with minimum cost, usually the number of test cases. It is not

possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Partitioning is one of the common techniques. If the input space is partitioned and all the input values in each partition are equivalent, then only one representative value in each partition needs to be tested to sufficiently cover the whole input space.

- **White box testing** uses the code of the program to derive test data. Hence for white box testing, the test cases cannot be determined until the code has actually been written. At first glance, white box testing seems unsafe. White box testing or structural testing cannot find errors of omission. However, requirements sometimes do not exist, and are rarely complete. This is especially true near the end of the product development time line when the requirements are updated less frequently and the software itself begins to take over the role of the specification. The difference between black box and white box testing blurs near release time.

- In recent years the term **Grey box testing** has come into common usage. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey-box because the input and output are clearly outside of the black-box we are calling the software under test. This is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test.

Tests can be categorized based on the parts of the code which are tested:

- **Unit testing** tests the minimal software component, or module. Each unit or module of the software is tested to verify that the design for the unit has been correctly implemented. In an object oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

- **Integration testing** test the interfaces and interaction between integrated units (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system. Integration can be performed with a big-bang (all at once), bottom-up (first low-level modules), top-down (start with the first level subsystems and use drivers) or sandwich (integrate groups of modules with other groups) approach.

- **System testing** tests a completely integrated system to verify that it meets its requirements.

- **System integration testing** verifies that a system is integrated to any external or third party systems defined in the system requirements.

- **Acceptance testing** is conducted by the end-user, customer, or client to validate whether or not to the product is "acceptable".

Some other testing types which are based on different testing purposes (instead of on phase in the software lifecycle) are:

- **Functional testing** tests at any level (class, module, interface, or system) for proper functionality as defined in the specification. Functional tests usually contain an expected result, which describes the expected output of the test. Based on the output the test is successful or not. Functional testing is the most common type of testing.

- **Performance testing** can be applied to understand the scalability of a program and to benchmark the performance of a program in different environments. This sort of testing is particularly useful to identify performance bottlenecks. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.
- **Regression testing** is similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of the software. Regression testing ensures reported software defects are not present in a new release. Though regression testing can be performed manually, an automated test suite is often used to reduce the time and resources needed to perform the required testing.
- **Stress testing** is conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails. A graceful degradation under peak load leading to non-catastrophic failure is the desired result. Often stress testing is performed using the same process as performance testing but employing a very high level of simulated load.

2.3 Test evaluation

The number of errors found by a test depends on 2 things: the quality of the software and the quality of the test. If a testing process does not detect any errors then either there are no errors in the software or the tests are not good enough. It is therefore important to control the quality of the testing process to obtain useful information about the quality of the software being tested.

2.3.1 Black box: specification-based

A black box test checks if certain functionality works as specified. The quality of a black box test suite is therefore high if all functions are tested according to their specification. Since we cannot select all input values for functions most of the time, the difficulty in generating black box tests is selecting appropriate representatives for a certain domain. Suppose that we would like to test the function *even*. This function uses only 1 argument, which is an integer. The function *even* returns true if the input is an even number and false if the input is an odd number. We cannot test the function for all integers, since there are to many possible input values. We have to select some representative values. It would be wise to test the function for some small and (very) large random values besides the boundaries of the integer type. This leads to the test set -2^{64}, −1234567, -12345, -5, -1, 0, 1, 2, 5, 12345, 1234567, 2^{64}-1.

2.3.2 White box: code coverage

When the internal structure of a software program is known, tests can created based on the structure of the program (instead of the requirements). The quality of a test case can also be measured in terms of the code. For this the term **code coverage** is used. Code coverage can be applied to different units in the code, for instance statements or functions. We distinguish the following different types of coverage:

- **Statement coverage** reports whether each executable statement is executed (at least once). If the statement coverage is 50%, then only half of the statements of the program is executed by the test suite. One argument in favor of statement coverage over other metrics is that bugs are evenly distributed through code; therefore, the percentage of the covered executable statements reflects the percentage

of faults discovered. However there are a lot of disadvantages of using statement coverage as a metric. Consider, for instance, the following C/C++ code fragment:

```
int *i;

if(boolean)
  i=malloc(sizeof(int));
i=5;
```

If all test cases only use the value *true* for the variable *boolean*, then the statement coverage for this program is 100%. However the code contains an error: if the variable *boolean* is false, then there is no memory allocated for the variable i and the program crashes when trying to assign the value 5 to i. Furthermore statement coverage does not report whether loops reach their termination condition. Since do-while loops always execute at least once, statement coverage considers them as non-branching statements and there are no further restrictions on the number of times the loop has to be executed. Statement coverage is completely insensitive to the logical operators ($\wedge$ and $\vee$) and it cannot distinguish consecutive switch labels.

- **Decision or branch coverage** This metric reports whether boolean expressions tested in control structures (such as the if-statement and while-statement) evaluated to both true and false. The entire boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Additionally, this metric includes coverage of switch-statement cases, exception handlers, and interrupt handlers. This metric has the advantage of simplicity without the problems of statement coverage. A disadvantage is that this metric ignores branches within boolean expressions which occur due to short-circuit operators (if the condition of an if-statement contains a function, branch coverage does not guarantee that the function is executed, only that the condition evaluates to both $true$ and $false$).

- **Condition coverage** Condition coverage reports whether each individual condition of a boolean expression is evaluated to both $true$ and $false$ (unlike decision coverage in which only the expression itself has to evaluate to $true$ and $false$). Therefore condition coverage has a better sensitivity to the control flow.

- **Multiple condition coverage** Multiple condition coverage reports whether every possible combination of boolean subexpressions occurs. As with condition coverage, the sub-expressions are separated by logical-and and logical-or, when present. So for a boolean expression with N subexpressions the number of combinations is 2^N. An advantage of multiple condition coverage is that it requires very thorough testing. A disadvantage of this metric is that it can be tedious to determine the minimum set of test cases required, especially for very complex boolean expressions. An additional disadvantage of this metric is that the number of test cases required could vary substantially among conditions that have similar complexity. Some of the combinations could be unfeasable, which could lead to less than 100 % coverage.

- **Path coverage** A path is a unique sequence of branches from the function entry to the exit. Path coverage reports whether each of the possible paths in each function has been followed. Since loops introduce an unbounded number of paths, this metric considers only a limited number of looping possibilities. A large number of variations of this metric exist to cope with loops. Boundary-interior path testing considers two possibilities for loops: zero repetitions and more than zero repetitions [30]. Path coverage has the advantage of requiring very thorough testing. Path coverage has two severe disadvantages. The first is that the number of paths is exponential to the number of branches. For example, a function containing 10 if-statements has 1024 paths to test. Adding just one more if-statement doubles the count to 2048. The second disadvantage is that many paths are impossible to follow due to impossible combinations of the boolean expressions.

Chapter 3

ATerm library

In this chapter we explain what the ATerm library is and why it was created. We look at the features of the library. Finally we discuss and analyze the tests and the available documentation.

3.1 History and motivation

ATerms (short for Annotated Terms) are an abstract data type designed for the exchange of tree-like data structures between distributed applications. ATerms and the ATerm library were designed and implemented in the beginning of 1998, independent of the Extensible Markup Language and OMGs Interface Definition Language (IDL) [34]. Typical applications who could use the ATerm library are parsers, type checkers, compilers, formatters, syntax-directed editors, and user-interfaces written in a variety of languages. An API generator (APIGEN) has also been developed, which allows type safe manipulation of ATerms without loss of efficiency and hence ensures an agile way of developing applications based on the ATerm library.

After an inventory was made of the term data types used in projects related to the ASF+SDF Meta-Environment [37], it turned out that 6 different term data types were used [36]. This gave rise to the idea of using only one data type for all different term data types. Though some implementations existed for exchanging data structures, like Microsoft's OLE [17], Java's serialization interface [21] and OMG's Interface Definition Language [23], none were *open, simple, efficient, concise*, and *language independent*, which are the necessary requirements for the exchange of complex data structures between distributed applications. A first implementation of the (new) general data type was used in the Toolbus implementation [12]. After the success of this implementation, ATerms [35] were introduced with a stand alone library and its implementation has been improved ever since, most notably parts related to the garbage collector [28] and the API [18]. Though first introduced only to serve a local problem (namely that of exchanging data between different tools connected to the Toolbus), ATerms have been used in a lot of other projects such as Stratego [11], mCRL [13] and the previously mentioned ASF+SDF Meta Environment.

3.2 Features

We will now give a brief overview of some of the features of the ATerm library. More details can be found in [35].

3.2.1 Data types

The data types of ATerms are defined as follows:

- INT: an integer constant (32-bits integer).
- REAL: a real constant (64-bits real).
- APPL: a function application consisting of a function symbol and zero or more ATerms (arguments). The number of arguments of the function is called the arity of the function.
- LIST: a list of zero or more ATerms.
- PLACEHOLDER: a placeholder term containing an ATerm representing the type of the placeholder.
- BLOB: a blob (Binary Large data OBject) containing a length indication and a byte array of arbitrary (possibly very large) binary data.
- a list of ATerm pairs may be associated with every ATerm representing a list of (label, annotation) pairs.

Each of these constructs except the last one (i.e. INT, REAL, APPL, LIST, PLACEHOLDER, and BLOB) form subtypes of the data type ATerm. These subtypes are needed when determining the type of an arbitrary ATerm. The last construct is the annotation construct, which makes it possible to annotate terms with transparent information (the result of most operations is independent of annotations). These annotations can thus be used for typesetting information in parse trees or type checking information in abstract syntax trees.

3.2.2 Level one and two interface

The current stable version of the library has the total of 315 functions, 12 C source files and over 15000 lines of code. The functionality is split between a level one and a level two interface. The level one interface contains the basic interface for handling ATerms, while the level two interface contains all the functions of the library. The operations in the level one interface fall into three categories: making and matching ATerms, reading and writing ATerms and annotating ATerms. The total of only 13 functions, which are all in the level one interface, provide enough functionality for most users to build simple applications with ATerms. Of these 13 functions 4 functions are used for making and matching ATerms, 6 functions are used for reading and writing ATerms and 3 functions are used for annotating ATerms.

Making and matching ATerms

The basic functions of the ATerm Library are the making ($ATmake$) and matching ($ATmatch$) of ATerms.

- make(compose) a new ATerm by providing a pattern for it and filling in the holes in the pattern.
- match(decompose) an existing ATerm by comparing it with a pattern and decompose it according to this pattern.

Patterns are just ATerms containing placeholders. These placeholders determine the places where ATerms must be substituted or matched. Here are some examples of the use of these functions:

```
ATerm term[4];
ATerm list[3];
ATerm appl[3];

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

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]);

/* 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);
```

3.2.3 Maximal subterm sharing

To minimize memory usage only *new* terms are created, i.e. terms that do not already exist. If the term, which is to be constructed, already exists, then the old term is reused, ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms and leads to maximal sharing of subterms. The library functions that construct terms make sure that shared terms are returned whenever possible. The sharing of terms is thus invisible to the library user.

Because of the maximal sharing, a check is performed to see if the term already exists whenever a term is created. A hash table is maintained for this purpose. The hash table does not contain the terms themselves, but pointers to the terms. Using the hash table has some advantages: the considerably reduced memory usage leads to reduced execution time. The equality check on terms also becomes very efficient. A consequence of maximal subterm sharing is that whenever a term is created, it cannot be destroyed since other terms may still refer to it. Hence no destructive updates on ATerms are allowed.

3.2.4 Automatic garbage collection

The C implementation of the ATerm library offers, in addition to maximal subterm sharing, automatic garbage collection. This frees the software developer from all the effort of explicitly allocating and deallocating terms. The garbage collector was initially based on the traditional mark-and-sweep garbage collector as developed by Boehm and Weiser [15]. However, one of the consequences of the maximal subterm sharing

is that destructive updates of terms are not allowed. This leads to the following invariant: the children of an ATerm are always older than the ATerm itself. Moreau and Zendra [28] have introduced a generational garbage collector in the ATerm library that exploits this invariant. The generational garbage collector is based on the assumption that newly created objects have on average a shorter life time than older objects. Older objects are thus inspected less frequently by the garbage collector. The observed efficiency gain for applications using the ATerm library is between 19 % and 35%.

3.2.5 Functions

The current stable version of the library consist a total of 315 functions divided over 12 C source files and over 15000 lines of code. Most functions are executed relatively independently (using only 1 or 2 other functions, which themselves can be executed in isolation). Examples of these functions are *ATgetLength* (getting the length of a list), *ATwriteToString* (writing an ATerm to a string) and *ATgetArity* (getting the arity of a function application). However there are some functions, for instance those of the garbage collector, which are quite complex and require a lot of interaction with other functions. Although there are only a few of these "complex" functions, they add a great deal to the overall complexity of the library.

3.3 Current test suite

We will now look at the current tests of the ATerm library. We used the 12/01/2007 version of the source code of the ATerm library, this is (at the moment of writing) the last stable version available from the SVN-server from CWI. This version contains its own test suite.

3.3.1 Available test cases

The C files used for creating the ATerm library (the files located in the *aterm* directory) contain about 15000 lines of code. In the directory *test* there is a file *stress.c* containing the tests. This file contains several functions which perform calculations and use functions of the ATerm library. A check is made at certain points in the calculation to examine if the functions of the ATerm library operate as expected. For this the function *test_assert* is used. An example of a test function in the *stress.c* file is:

```
void testDict(void)
{
  ATerm key[4];
  ATerm value[4];
  ATerm dict[4];

  key[0] = ATreadFromString("key-0");
  key[1] = ATreadFromString("key-1");
  key[2] = ATreadFromString("key-2");
  key[3] = ATreadFromString("key-3");

  value[0] = ATreadFromString("val-0");
  value[1] = ATreadFromString("val-1");
  value[2] = ATreadFromString("val-2");
  value[3] = ATreadFromString("val-3");
```

```
  value[0] = ATreadFromString("val-0");

  dict[0] = ATdictPut(ATdictCreate(), key[0], value[0]);
  dict[1] = ATdictPut(dict[0], key[1], value[1]);
  dict[2] = ATdictPut(dict[1], key[2], value[2]);
  dict[3] = ATdictPut(dict[2], key[3], value[3]);

  test_assert("dict", 1, ATdictGet(ATdictCreate(), key[0]) == NULL);
  test_assert("dict", 2, ATisEqual(ATdictGet(dict[1], key[0]), value[0]));
  test_assert("dict", 3, ATisEqual(ATdictGet(dict[2], key[1]), value[1]));
  test_assert("dict", 4, ATisEqual(dict[2], ATdictRemove(dict[3], key[3])));

  printf("dictionary tests ok.\n");
}
```

The first argument of the *test_assert* function is the suite to which the test belongs and the second argument is the sequence number of the test. The third argument is a boolean, if this boolean evaluates to $FALSE$, the test fails, a message is sent to the output stream and the execution is aborted. If the boolean evaluates to $TRUE$ the test was successful and the execution continues. In the above code some calculations are performed on a dictionary (values are added, retrieved and removed) and the function *test_assert* is used to check if the functions work as expected. In total the *stress.c* file contains 25 test functions, such as the *testDict* function, which each test a certain aspect of the library. A list of all the functions can be found in Appendix A.

There are also some other files which test some of the functionality of the library, though their intention is not to find bugs, but merely to illustrate the use of the library. These files are:

- *fib.c*: implementation of the Fibonacci principle
- *primes.c*: generation the first few prime numbers
- *randgen.c*: generation of a random ATerm
- *termstats.c*: deriving meta-data from an ATerm

3.3.2 Available documentation

Unfortunately there are no documents containing information (expected results, intention of the test, relation to previous bugs, etcetera) about the test cases. From the use of the *test_assert* function the expected results of the calculations are clear. However these results are specific to the input values, so the expected results for other input values require some extrapolation and reasoning (which is not always straightforward). The source code also contains a folder *doc* (besides the folder *test*), which has some documentation describing the working of the ATerm library. Apart from the manual, the folder contains papers describing the global working of the library (no details about the working of specific functions). Apart from some examples and notes on using the library the manual has a description of each function in the library. The manual however is outdated, since some functions which are not present are described in the manual (e.g. *ATinitialize*) and vica versa (e.g. *ATBinit*). The descriptions of the functions also are textual and somewhat ambiguous. Therefore the manual can only moderately be used for creating formal specifications of the functions.

3.3.3 Evaluation

We will now take a look at the quality of the current tests. The tests give a certain quality judgement about the library: does the library work as expected? Note that the test cases can have incorrect expected results, in which case the tests will fail even though the library works correctly. We assume that the current tests are correct. The tests themselves also can be evaluated. Do they test every aspect of the library? To answer this question, we have to use the source code of the library instead of the linked library and use a coverage tool to measure which parts of the source code are executed by the tests. We choose to create one file containing the source code of the entire library. This file was created manually, adhering to the different dependencies imposed by the inclusion of the files. Some functions had to be renamed to avoid name clashes also. Using this source file, the *stress.c* file was compiled using GCC [4] and the command line options -$fprofile\text{-}arcs$ and -$ftest\text{-}coverage$. These options are necessary to gather coverage information using GCOV [5] after execution. More information about these command line options and the GCOV tool can be found in section 5.3.1 and 5.4.2. Execution of the exectionable resulted in the following text:

```
Allocating 18 nodes of size 3:
Result: 0x6d30bc
Result: 0x6d30c8
Result: 0x6d30d4
Result: 0x6d30e0
Result: 0x6d30ec
Result: 0x6d30f8
Result: 0x6d3104
Result: 0x6d3110
Result: 0x6d311c
Result: 0x6d3128
Result: 0x6d3134
Result: 0x6d3140
Result: 0x6d314c
Result: 0x6d3158
Result: 0x6d3164
Result: 0x6d3170
Result: 0x6d317c
Result: 0x6d3188
test succeeded.
symmies[0]: application
symmies[1]: "application"
symmies[2]: "An \" \n \r \t \\ application"
symmies[3]: "application"
symmies[4]: application
symbol tests ok.
application tests ok.
list nodes: [...(0)], [...(1)], [...(2)], [...(3)]
result of ATremoveElement: [1,3,2]
list tests ok.
destr_false_count=4, destr_true_count=2
aint[0] = 1234
```

```
real[0] = 1.234567800000000e+00
<int>
"000000000009123456789"
AT_calcCoreSize(f(1,[2,3],<[a,b]>,1.243000000000000e+00,g(h(i(a,a),a),a,a))) = 412
AT_calcCoreSize(1) = 12
AT_calcCoreSize(f(1,2)) = 66
term read: []
term read: f(a)
term read: "f x"(b)
term read: "f \r \n \t \\ q"
term read: [a,b,c]
term read: f(a,b,1,2)
term read: <"just a test">
term read: f(a,b){[1,2]}
term read: 1
term read: -123
term read: 1.230000000000000e+00
term read: -1.230000000000000e+00
term read: -1.200000000000000e+04
term read: 1.230000000000000e+05
term read: 1.230000000000000e-03
term read: 9{[type,abcdefghi]}
term read: 0{[type,f{[a,1],[b,2],[c,3]}]}
term read: "abc"{[<just-a-test>,"what a test!"]}
term read: abc
term read: "the end"
read from string: f(1)
read from string: f
read from string: f(a,b,<123>,4.560000000000000e-01,"f")
read from string: []
read from string: f{[a,1],[b,ab{[1,2]}]}
read from string: <int>
read from string: "quoted: \"abc\""
Next term should give a parse error at line 0, col 17
dictionary tests ok.
annotation tests ok.
ATmake with 9 args: f(1,2,3,4,5,6,7,8,9)
ATmakeAppl: f(1,2,3,4,5,6,7,8,9)
The following two tests should generate parse errors.
readFromTextFile: parse error at line 0, col 17:
f("Error",[2,3] [
ATreadFromString: parse error at or near:
<int
    ^
ATreadFromString: parse error at or near:
f(<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,
```

```
<int>,<int>,<int>,<int>,<int>,<int>,<int>asdfaksdjfhasjkhf)
                                              ^
make tests ok.
match tests ok.
term written to binary string: f(1,a,<abc>,[24,g]{[a,b]}), size=110
term read from binary string : f(1,a,<abc>,[24,g]{[a,b]})
baffle tests ok.
taf tests ok.
gc tests ok.
mark tests ok.
table tests ok.
indexedSet tests ok.
TB legacy tests ok.
checksum tests ok.
diff tests ok.
compare tests ok.
```

In total it took about 3 seconds for the test file to execute. The above text implies the tests were successful and the parts of the library that were tested, operate as expected. Using the GCOV tool we got the following results for the different types of coverage:

- Line/Statement Coverage: **76,41**%
- Branch Coverage: **54,53**%
- Calls executed: **61,87**%

So about 25% of the source code of the library is not tested at all! Of the **315** functions which are defined in the library, **49** (both internal and external functions) were not executed at all.

3.3.4 Conclusion

The current "test file" is a nice start for creating a good test suite for the ATerm library. It has a lot of examples on how to use some of the functions of the library and what they should do. However a big part of the code is not covered and a lot of functions are not tested. Therefore additional tests have to be created to test these parts of the library. Since the current tests have no documentation, we have to create some form of documentation for the test suite as well. Having the documentation makes it easier to change the tests (if the source code of the library is changed) and to document their purpose, expected results, etcetera. The main problem is that (new) tests have to be created and specified without much documentation about the code, the current test cases, or their expected results and purpose.

Chapter 4

Testing based on universal quantifications

In the previous chapter we discussed some of the shortcomings of the test suite of the ATerm library. Apart from the fact that 49 functions were not tested, the tests are not documented and cannot be changed easily if functionality is added to the library or the output value of a function is changed. We have some specific requirements for the testing framework, which we discuss in section 4.1. In section 4.2 we look at the frameworks we could use for maintaining a test suite. We desribe a specification language, a modified version of universal quantifications in section 4.3.2.

4.1 Requirements

Since we want to compare different versions of the ATerm library, we will use a black box testing framework. We cannot use the knowledge of the structure of the code, since this might change with the newer versions. Considering the size of the library, the only realistic option is an automated approach. Using an automated tool also allows us to test new versions of the library quickly. Since we have no documentation available, we want to create the requirements/specification for the functions easily and based on them create the test cases (automatically).

Most of the functions of the ATerm library can be specified quite easily in terms of pre- and postcondition. For instance if we use the function *ATappend* to add a term to the end of a list, we know that the size of the resulting list is 1 bigger than the size of the original list. The resulting list also contains the same elements as the original list, only the last element is the appended term. For testing these kinds of functions, we can use an unit testing framework to administer the results of the tests. There are a quite a lot of (C) unit testing frameworks of which CUnit [2] is the best known. However a function like *ATprotect*, which protects a term from being deleted by the garbage collector, cannot be specified easily in terms of pre- and postcondition. We want to model that for an arbitrary term, if we execute *ATprotect* on that term followed by the execution of *AT_collect*(), which performs a garbage collection cycle, the term is still valid (the term is not deleted by the garbage collector) and has its original value. The function *ATprotect* cannot be specified in isolation, but if we use the function in combination with other functions, we know exactly what the result should be. We want to use a testing framework which has the flexibility to easily specify a function like *ATappend*, but also the ability to model the use of the *ATprotect* function.

4.2 Existing testing tools

4.2.1 AsmL

AsmL is the Abstract State Machine Language [3]. It is an executable specification language based on the theory of Abstract State Machines. The current version, AsmL 2 (AsmL for Microsoft .NET), is embedded into Microsoft Word and Microsoft Visual Studio.NET. It uses XML and Word for literate specifications. There is a test generation tool which works with this specification language, and Microsoft researchers have reported on its characteristics in [22]. Their test generation algorithm achieves total transition coverage of the derived finite state machine.

4.2.2 TGV/CADP

TGV (Test Generation with Verification) is a test case generator developed at the IRISA and VERIMAG laboratories. It accepts input in several specifications languages including SDL [16]. The output of the test generator is in the TTCN format, and contains the expected results of the test. The test generation directives may be in the form of FSM models of the test purposes. The tool has been used extensively in experimental and industrial situations, mainly in the telecommunications industry. The test generator is incorporated in the CADP package and may be downloaded from the website at [1].

4.2.3 TorX

The TorX [10] is an architecture for test generation and execution from the University of Twente. Within this architecture there is a test generator which accepts test purposes in a similar format to TGV. The modeling languages supported by TorX are LOTOS [38], PROMELA [9], and SDL [16].

4.2.4 RESOLVE

RESOLVE [39] is a specification language which can be used to specify the functional behavior of a software component. This component can be a class, but also an entire Java Bean. In essence, each software component provides a simple "hook" interface (with no run-time overhead) that can be used in adorning the component with sophisticated built in test capabilities. Sophisticated "decorator" components (wrappers) that provide a number of selfchecking and self-testing features can then be used to encase the underlying component. Several frameworks exist which use a RESOLVE specification for a class to generate test cases, the best known is made by Edwards [19].

4.2.5 ESTELLE

ESTELLE [6] is formal description technique based on an extended state transition model. A framework proposed by Do Y. Lee and Jai Y. Lee [26] uses a restricted form of ESTELLE to directly derive a single reduced finite state machine, which is called a Control Flow Graph (CFG), from the specification. The derived CFG provides a basis for the automatic test case generation.

4.2.6 Conclusion

We have discussed several different frameworks which use a model as a basis for generating test cases. However these frameworks either are to complex to properly specify functions like *ATappend* (AsmL) or do not provide the necessary constructs to specify a function like *ATprotect* (RESOLVE). Therefore we choose to create our own testing framework which allows us to handle both cases properly and easily.

4.3 Specification language: universal quantifications

As mentioned in the previous section we want to create a framework in which we can easily create a specification for a function and generate test cases based on that specification. For the specification language, we use the language of universal quantifications, which we explain in the next few sections.

4.3.1 Predicates

A predicate is a symbolic expression that evaluates to either true or false, based on the arguments of the predicate. Predicate symbols are used to denote the expressions and the arguments are written between brackets. For example if we define P to be "is bigger than 5", then $P(3)$ means "3 is bigger than 5" (which is false) and $P(7)$ means "7 is bigger than 5"(which is true). $P(x)$ is equal to "x is bigger than 5", the truth value depends on the definition of the variable x.

4.3.2 Universal quantifications

Suppose we define the predicate $R(x)$ to be "x+x is equal to 2*x". For every natural number, this predicate is true. When we want to express this, we can write $R(0) \wedge R(1) \wedge R(2) \wedge ... \wedge R(n)$. The dots represent the predicates for all other natural numbers and n is the last natural number. A shorthand notation for this expression is ($\forall$ x: x $\in \mathbb{N}$: R(x)). This expression is pronounced as "For every natural number, R holds". Such a statement is called a **universal quantification**. In the part after the $\forall$ we specify what variables are bound to a certain domain. This "restriction" to the use of a variable (the scope of the variable) ends as soon as the quantification is closed. In the example, the bound variable is x. The second part (between the two colons) of such a quantification is called the **range** of the quantification. The range specifies what values are used for the bound variables (the domain of the variable). In the example the bounded variable x ranges over the set of natural numbers. The third part of an universal quantification contains a predicate, which typically has all the bounded variables as arguments (this is not mandatory, but it is unnecessary to use a bounded variable if it is not used in a predicate).

Note that besides the universal quantification, also an existential quantification exists. Similar to the universal quantification being equivalent to the conjunction of a number of predicates, the existential quantification is the disjunction of a number of predicates. For this type of quantification the symbol $\exists$ is used. So using the predicate R as above, ($\exists$ x: x $\in \mathbb{N}$: R(x)) is equal to $R(0) \vee R(1) \vee R(2) \vee ... \vee R(n)$. Both the universal and the existential quantification are widely used in predicate logic.

Some examples of universal quantifications are:

($\forall$ n: n $\in \mathbb{N}$: n $\times$ n == n^2)
Description: "Multiplying a natural number with the same number is the same as the square of the number." This expression evaluates to $TRUE$.

(∀ i: i ∈ ℤ: i > 0)
Description: "All integers are bigger then 0." This expression evaluates to $FALSE$ since 0 is an integer and it is not bigger than 0.

(∀ r: r ∈ ℝ: r + 4.2 > r)
Description: "If you add 4.2 to a real number, then the result is bigger than the original real number." This expression evaluates to $TRUE$.

In these examples we used the shorthand notation ℕ, ℤ and ℝ for the domain of natural numbers, integers and real numbers respectively. The bounded variable (n, i and r respectively) also does not occur in the description of the quantification. We could have used different names for these variables and the meaning of the quantification would be the same. We can also use multiple bounded variables in an universal quantification:

(∀ i, j: i ∈ ℤ ∧ j ∈ ℤ: i * j ≥ 0)
Description: "The multiplication of 2 integers is always bigger than or equal to 0." This expression evaluates to $FALSE$, since the multiplication of a negative integer with a positive integer results in a number smaller than zero. Note that the different variables are separated by a comma and the domain specifications are separated by a logical and (∧).

Since universal quantifications are boolean expression themselves, we can use them as predicates in another universal quantification:

(∀ i: i ∈ ℤ: (∀ j: j ∈ ℤ: (i > j) ⇒ (i+1 > j+1)))
Description: "If an integer is bigger than an other integer then the result of adding 1 to the first integer is bigger than the result of adding 1 to the second integer." This expression evaluates to $TRUE$.

4.3.3 Code snippets

As mentioned at the start of this chapter, we want to be able to specify the result of certain execution sequences. For instance we want to specify that a protected term is valid after a garbage collection cycle. Using the current syntax of an universal quantification, we are not able to do this. Therefore we extend the syntax of an universal quantification with **code snippets**. A code snippet contains source code, which is executed before the predicates of the quantification are evaluated. When such a code snippet is used, the programming language of the source code also has to be specified since different languages can have different syntax. Code snippets may occur in the range of a quantification. Consider for instance this quantification (assuming the programming language is C):

(∀ i: i ∈ ℤ ∧ [[init();]]: check(i))
Description: "If you perform the *init* function, then for every integer the function *check* applied to that integer evaluates to $TRUE$". Both the *init* and the *check* function have to be defined in order for this quantification to have any meaning. The only restriction we put on the code snippets is that code is correct for the specified programming language. Other than that, any (correct) code is allowed. We will elaborate more on the restrictions of these code snippets in section 4.3.4.

Some uses of code snippets (using the C programming language for the code snippets):

(∀ i: i ∈ ℤ ∧ [[int j=i*2;]]: j==i+i)
Description: "For every integer, if you create a new variable which is twice the original integer, then the variable is equal to the sum of the original integer with again the original integer".

(∀ i: i ∈ ℤ ∧ [[int j; foo(i,&j);]]: j==5)
Description: "If you use the foo function with arguments an arbitrary integer and a pointer to an integer variable j, then j is equal to 5".

The function *ATprotect* can now be specified as followed:

(∀ t: t ∈ $Term$ ∧ [[ATprotect(t);AT_collect()]]: ATisvalidTerm(t))
Description: "If you protect a term and perform a garbage collection cycle, then the term is still valid". The domain $Term$ is used to indicate the collection of all Terms.

4.3.4 Grammar

By adding the code snippets we have created a new formal language. We will now use a context-free grammar [33] for a formal specification of the language. A context-free grammar is a quadruplet $\langle N, T, s, P\rangle$. N and T are the disjoint sets of nonterminals and terminals. $s \in N$ is called the start symbol. P is a finite set of production rules with $P \subset N \times (N \cup T)^*$. A production rule $p \in P$ is a tuple $\langle l, r\rangle$ with $l \in N$ and $r \in (N \cup T)^*$. p is usually written as $l \rightarrow r$. In our case N contains $Quantification$, $Declarations$, $Declaration$, $Predicates$ and $Predicate$. N also contains INT, VAR, $DOMAIN$, $TYPE$, $CODE$ and $BOOLEAN$, but we will not present production rules for those nonterminals. INT simply is an integer, VAR is a name for a variable, $DOMAIN$ is the name of a domain/set of values of a variable, $TYPE$ is a valid type, $CODE$ is source code and $BOOLEAN$ is a boolean expression. Production rules for these nonterminals can be found in other papers, we want to focus on the nonterminals relevant for the universal quantifications. The terminals are "(", "∀", ": ", ")", " ∧ ", "≤", "<", " in ", "[[", "]]", their use will become apparent from the production rules. The start symbol is $Predicate$ and the production rules are:

[quantification]	Quantification	→	"(∀" Declarations " : " Predicates ")"
[declarations-0]	Declarations	→	Declaration
[declarations-1]	Declarations	→	Declaration " ∧ " Declarations
[declaration-0]	Declaration	→	INT "≤" VAR "<" INT
[declaration-1]	Declaration	→	VAR " in " DOMAIN
[declaration-2]	Declaration	→	"(" TYPE ")" VAR " in " DOMAIN
[declaration-3]	Declaration	→	"[[" CODE "]]"
[predicates-0]	Predicates	→	Predicate
[predicates-1]	Predicates	→	Predicate " ∧ " Predicates
[predicate-0]	Predicate	→	BOOLEAN
[predicate-1]	Predicate	→	Quantification

Note that we have made a modification to the previous notion of universal quantification: a variable is bound implicit by the declaration. So instead of

($\forall$ i: i $\in \mathbb{Z}$: i+1>0)

We use the expression

($\forall$: i $\in \mathbb{Z}$: i+1>0)

We also introduced the use of typecasts. If no typecast is used, we assume the variable to be a string, since that type is used most often. Using explicit typecasts makes it easier to reuse the values in a domain and to declare the variable in the implementation. Since we can reuse the *List* domain for both lists and ATerms and the following quantification is valid:

($\forall$: (ATermList)l $\in$ List $\wedge$ (ATerm)t $\in$ List: G(l,t))

Restrictions

There are some restrictions to the use of these production rules. We will not give a formal specification of these restrictions, but simply illustrate the use of the rules by example.

- **Variable names can be bounded only once**. If a variable is bound using an universal quantification, a variable with the same name cannot be bound again in the other declarations or in the predicate. The following quantifications are therefore illegal:

  ($\forall$: i $\in \mathbb{Z} \wedge$ i $\in \mathbb{N}$: i+1>0)
  ($\forall$: i $\in \mathbb{Z} \wedge$ i<10 : i<20)
  ($\forall$: i $\in \mathbb{Z}$: ($\forall$: i $\in \mathbb{N}$: i+1>0))

  However the following expression is correct:

  ($\forall$: i $\in \mathbb{Z}$: i+1>0) $\wedge$ ($\forall$: i $\in \mathbb{N}$: i>0)

  The scope of the first bounded variable i ends at the end of the the first universal quantification. Hence i can be used as a bounded variabele in the second quantification.

- **Code snippets must contain correct code**. The code in the code snippets must be correct for the quantification to have any meaning. If the code is used in combination with the bounded variables, then that code can be compiled and executed without any errors. The following quantifications are therefore not allowed (when chosing C as programming language for the code snippets):

  ($\forall$: i $\in \mathbb{Z} \wedge$ [[int j=foo(i);]] : j==7)

  This quantification is incorrect if the function foo is not defined, not declared or has compile errors. The foo function also must have 1 integer argument (the type of the argument can be derived from the domain of the variable). The following quantifications are also incorrect since the contain incorrect C code:

(∀: i ∈ ℤ ∧ [[int j=7.7;]] : j==7)
7.7 *is not an integer value*

(∀: i ∈ ℤ ∧ [[j=7;]] : j==7)
The variable j is not declared

(∀: i ∈ ℤ ∧ [[int j=7 int k=7]] : j==7)
The semicolon is missing between the two statements

(∀: i ∈ ℤ ∧ [[int j; int k=j*5;]] : i>0)
Since j does not have a value, the computation of $j * 5$ *results in an error*

- **Variables and functions used in a boolean expression must be defined correctly in the preceding code snippets**. Again the scope of the bounded variables is important. The following quantifications are incorrect:

  (∀: i ∈ ℤ : j==7)
  j is not declared

  (∀: i ∈ ℤ ∧ [[int j;]] : j==7)
  j is not initialized

  (∀: i ∈ ℤ ∧ [[int foo(int j) return 5 ;]] : foo(i)==8.9)
  The result of the function foo is an integer and not a real

4.4 Generating test cases

We can use our (adapted) version of the universal quantification as a basis for generating test cases. Recall that (∀: i ∈ ℤ : P(i)) is equal to $P(0) \wedge P(1) \wedge ... \wedge P(n)$. So to *test* if the quantification holds (evaluates to true), we have to test the predicate for every single value of the domain(s) of the bounded variable(s). Since we have introduced the code snippets, we have to execute that code for every value of the variable(s) before we test the predicate for that value. If a quantification has 1 bounded variable and the domain of that variable has 10 values, then the quantification is tested by 10 test cases, one for each of the values. So if there are 3 bounded variables, with 10 values each, the quantification is tested by 10^3 test cases. Using the quantification as a basis has some advantages. The quantifications themselves are a compact specification method. With one quantification we document the expected outcome/calculation of a function for *every* input value. We can reuse code created by others for specific cases as well. The test file (as mentioned in the previous chapter) for instance contains the code:

```
Symbol sym;

sym = ATmakeSymbol("application", 3, ATfalse);

test_assert("symbol",1,ATgetArity(sym)==3);
```

We can reuse this code and create the following quantification:

(∀: i ∈ ℤ∧ [[Symbol sym=ATmakeSymbol("application",i,ATfalse);]]: ATgetArity(sym)==i)

Using only this one quantification, we can create a test case for every input value.

4.4.1 Domain limitations

When using the quantifications to generate test cases, we generate a test for every input value. However if we use the integer domain for a bounded variable, generating the test cases would lead to an infinite number of tests. Therefore we have to define our own domains, containing representative input values. As mentioned in chapter two, for the integer domain representative values would be -2^{64}, −1234567, -12345, -5, -1, 0, 1, 2, 5, 12345, 1234567, 2^{64}-1. For example we define Int to be the set of all these representative values for the integer domain. When using the quantifications, we assume that the quantification using the integer domain holds if the quantification using Int holds:

(∀: i ∈ Int: $P(i)$) ⇒ (∀: i ∈ ℤ: $P(i)$)

So, if we use these quantifications to generate test cases, we have to select representative values for every domain used in the quantifications. We have to select those values in a way such that we have confindence in the quantification to hold, when the predicate of the quantification holds for every representative value.

Chapter 5

Implementation: UQT

In the previous chapter we have introduced a formal language which allows us to use universal quantifications to specify functions. In this chapter we describe a framework which uses universal quantifications as input to create test cases. The implementation was made in C and compiled using GCC [4].

5.1 Overview

We have used, as depited in Figure 5.1 a three step approach for our testing tool: **UQT (Universal Quantification Tool)**. In step one the test cases are generated based on the quantifications. A source file is also created for the library that is being tested. The created source file can be reused when the quantification file is modified, so this functionality is usually used only once. In the second step, the test cases are executed with the CUnit library to create a test report and coverage files. A data file is used to get representative values for the domains used in the quantifications. Finally, the coverage files are used by GCOV to create coverage data and generate some GCOV files.

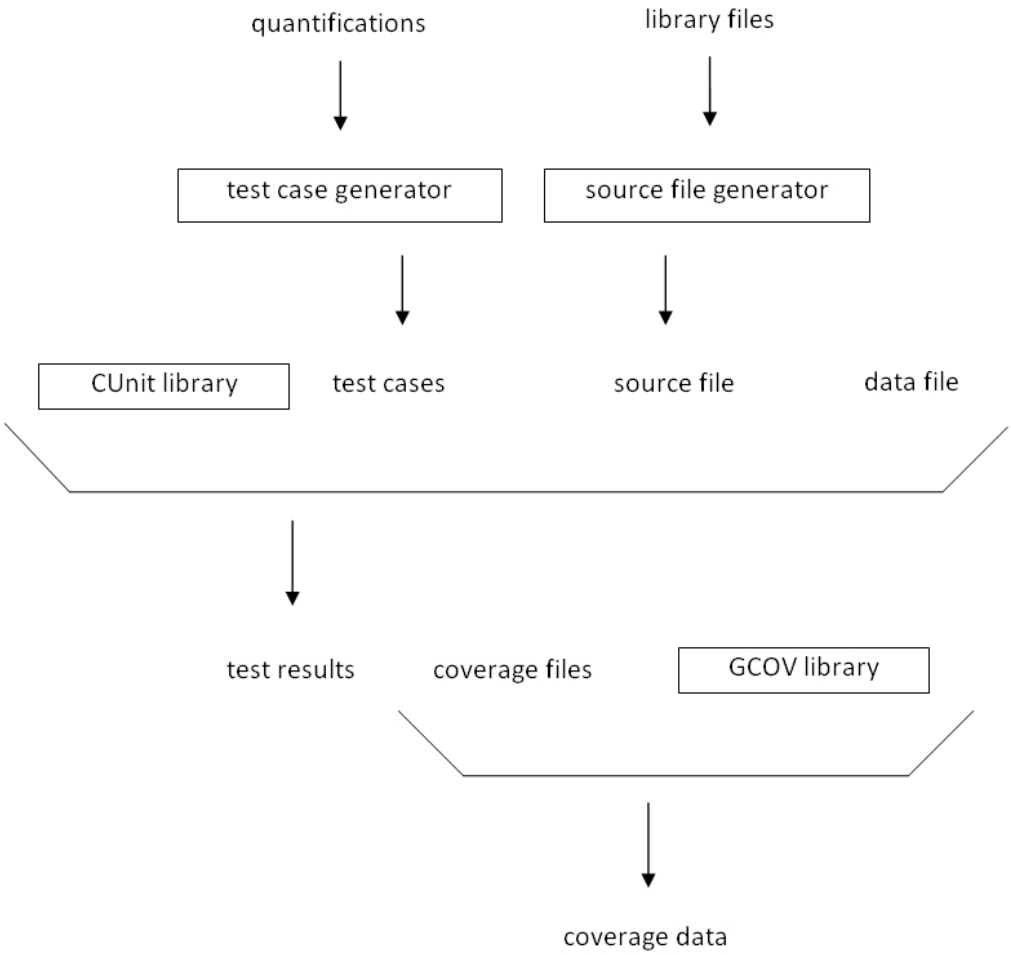


Figure 5.1: Overview of UQT

5.2 Creating the tests

We want UQT to be able to calculate if a set of quantifications hold. We also want to know which parts of the library are executed by checking those quantifications. To get coverage data, we need to execute the tests on the source code of the library, not on the linked library. In the next section we will describe an algorithm which creates a source file of a library. This file can then be used instead of the library. In section 5.2.2 we will describe how the tool uses the set of quantifications to generate the test cases.

5.2.1 Creating library source file

In the current version of the ATerm library the source code is divided over 12 C files and 20 header files. A lot of the C files use some of the other files. Therefore we cannot create the source file of the library by arbitrarily copy-pasting all the files into in file. We have to adhere to the order of inclusion: if $file1.c$ includes $file2.c$ then the code of $file2.c$ has to be placed before the code of $file1.c$ in the source file. Hence we used the following algorithm for creating a source file for a library:

Algorithm 1 Generate a source file for a library

$N \rightarrow 0$
$n \rightarrow 0$
$files \rightarrow [\ ]$

define $storeIncludes(filename)$:
for each file f included in $filename$ **do**
  if f does not occur in $files$ **then**
    $files[N] \rightarrow f$
    $N \rightarrow N + 1$
  else if f is contained in one of the first n elements of the array **then**
    $p \rightarrow$ *the position of* f *in* $files$
    remove f *from* $files$
    inserted f *into* $files$ *at position* p
    $n \rightarrow p$
  end if
end for
$n \rightarrow n + 1$

$storeIncludes(initialfilename)$
while $n \neq N$ **do**
  $storeIncludes(functions[n])$
end while

All the files which are included are stored in the array $files$. The total number of files stored in $files$ is equal to N and n contains an index such that the following property holds:

($\forall$ i : i $\in \mathbb{Z} \wedge$ 0≤i<n : ($\forall$ j : j $\in \mathbb{Z} \wedge$ 0≤j<i : *The* $file$ *named* files[i] *does not include the* $file$ *named* files[j]))
Description: All the files in the array uptil n do not include a file with a smaller index.

So if the algorithm terminates the following property holds (since for the *while* loop to terminate n must be equal to N):

($\forall$ i : i $\in \mathbb{Z} \wedge$ 0$\leq$i<N : ($\forall$ j : j $\in \mathbb{Z} \wedge$ 0$\leq$j<i : *The file named* files[i] *does not include the file named* files[j]))
Description: All the files in the array do not include a files with a smaller index.

Since the files do not include files with a smaller index, they only include files with a bigger index. Hence the last file in the array does not include any of the files in the array (since there is no bigger index). The file right before the last one, only includes the last file, etcetera. Hence if we create a source file based on the *reversed* order of the *files* array, we create a file which took the dependencies into account opposed by the inclusion of files.

5.2.2 Quantification file

The quantifications will be stored in one text file. The symbols "FA", "in", "<=" and "/\" will be used for the symbols $\forall, \in, \leq$ and $\wedge$. So for instance the quantification

($\forall$: i $\in$ Int $\wedge$ 0$\leq$j<10: R(i,j))

Will be written in the text file as

(FA: i in Int /\ 0<=j<10: R(i,j))

A parser was made from scratch for handling the quantifications. UQT will use the quantifications to generate a C file which can be executed using the CUnit library [2]. Execution of this file will result in an XML file containing the test results.

Test cases

Every quantification is used to create a test case, which all are added to one test suite. For every bounded variable, a C variable is declared depending on the typecast of that variable. If no typecast is used, the variable is declared as an ATerm and not a string. The reason we do this is that the quantifications we created contain a lot more ATerms than strings and hence the list of quantifications becomes more clear. Depending on the type a different parse function is used to assign a value to the variable. Using a *while* loop the different values are parsed into the variable. The predicates (those which are not an universal quantification, since the quantifications have to be treated recursively) will be tested using the CUnit function CU_ASSERT, which tests if a boolean expression holds and registers the result with the current test suite. Figures 5.2 to 5.5 give some examples of how the code is generated.

For every bounded variable in the quantification, a variable with the same name is used in the code. An additional variable is used for the linked list containing the representative values of the domain of the corresponding bounded variable. The name of this additional variable is the name of the bounded variable with "_list" added to it. The values are added to the linked list using the function *getDomainData*, which besides the domain and the name of the linked list also has the name of a text file as argument, which contains the representative values. We will explain this file in more detail in section 5.3.3. Using a while loop all the representative values are eventually parsed into the appropriate variable.

Quantification:

(∀: (int)i ∈ Int: R(i))

C code:

```
void testcase1() {
  struct node *i_list;
  i_list=NULL; append(&i_list,"start");
  getDomainData("Int","data.cps",i_list);
  while (i_list->link != NULL) {
    i_list=i_list->link;
    int i=atoi(i_list->data);
    CU_ASSERT(R(i,j))
  };
}
```

Figure 5.2: Example translation 1

Quantification:

(∀: (int)i ∈ Int ∧ 0≤j<10: P(i,j))

C code:

```
void testcase2() {
  struct node *i_list;
  i_list=NULL; append(&i_list,"start");
  getDomainData("Int","data.cps",i_list);
  while (i_list->link != NULL) {
    i_list=i_list->link;
    int i=atoi(i_list->data);

    int j;
    for(j=0;j<10;j++) {
      CU_ASSERT(P(i,j))
    };
  };
}
```

Figure 5.3: Example translation 2

Quantification:

(∀: (int)i ∈ Int ∧ [[int res=foo(i)]]: res==6)

C code:

```
void testcase3() {
  struct node *i_list;
  i_list=NULL; append(&i_list,"start");
  getDomainData("Int","data.cps",i_list);
  while (i_list->link != NULL) {
    i_list=i_list->link;
    int i=atoi(i_list->data);

    int res=foo(i);

    CU_ASSERT(res==6)
  };
}
```

Figure 5.4: Example translation 3

General code

So far we have described how for each quantification a test function is generated. However each of these functions has to be added to a test suite, a reference has to be made to the source file of the library and the header files of the CUnit framework have to be included. Appendix B contains an example C file with all the correct references and CUnit functions. The functions *init_suite_success* and *clean_suite_success* are used at the start and end of each test case. Using a dummy ATerm, the ATerm library is initialized. We use only one test suite for all the test cases and add each test case to that test suite. Some setup functions are executed before the functions *CU_automated_run_tests* and *CU_list_tests_to_file* are executed to run the tests and store the results in an XML file.

5.3 Executing the tests

The result of the first run are two files: a source file of a library and a file which will perform the tests using that source file. In this section we will describe what compile options are used when executing the test file, how UQT uses CUnit and how the representative values are stored in the data file.

5.3.1 Compile options

In order to use GCOV we have to compile the test file with GCC with the command line options *-fprofile-arcs* and *-ftest-coverage*. These options cause the compiler to insert additional code in the object files. These options also generate two files a *.bb* file and a *.bbg* file. The *.bb* file contains a list of source files (including headers), functions within those files, and line numbers corresponding to each basic block in the source file. The *.bbg* file contains a list of the program flow arcs for each function which in combination with

Quantification:

(∀: (ATermList)l ∈ List ∧ (ATerm)t ∈ Term ∧ [[ATermList l2=ATappend(l,t);]]:
(∀: 0≤j<ATgetLength(l): ATisEqual(ATelementAt(l,j),ATelementAt(l2,j)))
∧ ATisEqual(ATelementAt(l2,ATgetLength(l2)-1),t))

C code:

```
void testcase4() {
  l_list=NULL; append(&l_list,"start");
  getDomainData("List","data.cps",l_list);
  while (l_list->link != NULL) {
    l_list=l_list->link;
    ATermList l=ATparse(l_list->data);

    struct node *t_list;
    t_list=NULL; append(&t_list,"start");
    getDomainData("Term","data.cps",t_list);
    while (t_list->link != NULL) {
      t_list=t_list->link;
      ATerm t=ATparse(t_list->data);

      ATermList l2=ATappend(l,t);

      int j;
      for(j=0;j<ATgetLength(l);j++) {
        CU_ASSERT(ATisEqual(ATelementAt(l,j),ATelementAt(l2,j)))
      };

      CU_ASSERT(ATisEqual(ATelementAt(l2,ATgetLength(l2)-1),t))
    }
  }
}
```

Figure 5.5: Example translation 4

```
Term: qq(0,54.3,[1,3.4]) | [[]] | [1,4,5] | <real> | 6.66
 | "000000000005"abcd"" | 00000009:abcdefghi
Int: 1 | 5 | 6
Real: 1.2345 | 5.501
List: [1] | [[],[]] | [[]] | [a,b,c] | [1,suc(1),suc(suc(1))]

USymbol: aa | bb
QSymbol: "application" | "An \" \n \r \t \\ application"

Blob: "abkjjfgjjerjnghksdfgsdfgshosdghujksjngbiwruoijgpkmthuiodjkgpkeihfgjkjwuok3u9024sc"
 | "a" | "bbb"
```

Figure 5.6: Example data file

the *.bb* file enables GCOV to reconstruct the program flow. Upon executing a GCOV enabled user program, it dumps the counter information into a *.da* file at the time of exit.

5.3.2 CUnit

The CUnit library is used to create an XML file containing the test results. In order to use CUnit, we have to link the library to the test file at compile time. The test results include the number of the test cases, the number of successful test suites, test cases and assertions and also the boolean expression of all the assertions that evaluated to false. Using a markup file, we can see a nice file containing the test results. We will discuss this file in the next section.

5.3.3 Data file

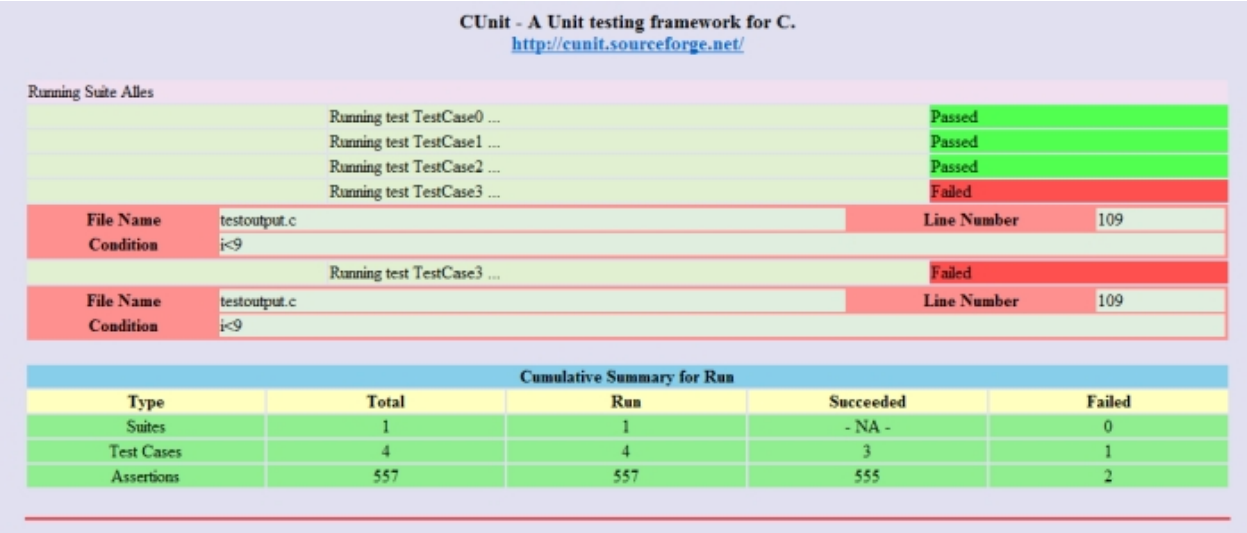
The representative values for every domain are stored in a linked list using the function *getDomainData*. These values are gathered by UQT from a data file. This data file contains several domain specifications: the name of a domain followed by a number of values. The domain name is followed by a ':', the values are separated by a '|'. The values for these domains can be chosen independently of the source code and in complete isolation of the library. Figure 5.6 contains an example data file.

5.4 Gathering results

After the test file is executed some files are created by the different tools. CUnit produces an XML file and GCOV produces coverage files. We will discuss these files in the next sections.

5.4.1 Test results

The result of executing the test file with CUnit is a XML file. The XML file can be opened using a stylesheet which is delivered with the CUnit framework. Figure 5.7 contains an example report.



CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Running Suite Alles

	Running test TestCase0 ...		Passed
	Running test TestCase1 ...		Passed
	Running test TestCase2 ...		Passed
	Running test TestCase3 ...		Failed
File Name	testoutput.c	Line Number	109
Condition	i<9		
	Running test TestCase3 ...		Failed
File Name	testoutput.c	Line Number	109
Condition	i<9		

Cumulative Summary for Run

Type	Total	Run	Succeeded	Failed
Suites	1	1	- NA -	0
Test Cases	4	4	3	1
Assertions	557	557	555	2

Figure 5.7: Example CUnit report

5.4.2 GCOV

Running GCOV with the program's source file name as argument, will combine the information from the *.bb*, *.bbg* and *.da* files and produce a listing of the code along with the number of times each line is executed. '######' Implies 0 executions. GCOV can also be run with some command line options, we will use $-b$ to get branch coverage data besides the statement coverage data. Figure 5.8 contains an example GCOV file.

Since the values 0 through 10 are used, the for statement is executed 11 times. The last time it is executed, the condition is false, so the statement in the body is executed 10 times (and that branch is executed 10/11=91% of the time). Since the sum of the numbers 0 through 9 is 45, the first guard is never true and the corresponding statement is not executed. When executing GCOV on the test file, coverage information is also displayed on the screen. Figure 5.9 contains the displayed text for the program from Figure 5.8 (source.c.gcov contains the GCOV code). In total there are 4 branches: 2 for the for statement and 2 for the if stamenent. Since both statements are executed, 'Branches executed' results in 100%. However since the first branch of the if statement is never executed, only 3 branches we taken at least once. Hence this metric has the value 75%.

5.5 Applying UQT to the ATerm library

In the previous sections we have discussed the framework we implemented which uses universal quantifications as a basis for generating tests: UQT. We used UQT to test the ATerm library; we created some quantifications which specify the functions of the library.

Using the *stress.c* file and the manual, we created a total of 88 quantifications specifying the working of the ATerm library, which can be found in Appendix C. We also created a data file containing values for the domains. We choose those values without the use of any tool or formalism, only using our common sense.

```
function main called 1 returned 100% blocks executed 78%
        1:    1:int main() {
call    0 returned 100%
        1:    2: int i, total;
        1:    3: total = 0;
       11:    4: for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
       10:    5: total += i;
        1:    6: if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:    7:  printf ("Failure\n");
call    0 never executed
        -:    8: else
        1:    9:  printf ("Success\n");
call    0 returned 100%
        -:   10:
        -:   11:}
```

Figure 5.8: GCOV example

```
File 'source.c'
Lines executed:87.50% of 8
Branches executed:100.00% of 4
Taken at least once 75.00% of 4
Calls executed:66.67% of 3
source.c:creating 'source.c.gcov'
```

Figure 5.9: GCOV example display

In the future we will spend more effort on choosing these values. The data file can be found in Appendix D. The 88 quantifications resulted in 88 test functions and 113938 assertions (compared to the 3205 assertions in the *stress.c* file). In total executing the test file took 40 seconds (which is quite reasonable compared to the number of assertions). All the assertions evaluated to true, only after removing the following quantification, which caused the system to crash:

(∀: l ∈ List ∧ [[ATermList l2=ATsort(l,ATcompare);]]: (∀: 1≤i<ATgetLength(l):
ATcompare(ATelementAt(l2,i-1),ATelementAt(l2,i))≤0))

We took a closer look at the function *ATcompare*, which is a function returning the result of a comparison between two ATerms. If they are equal, the result is 0, otherwise the result is either -1 or 1 depending on the types and values of the terms. The definition of *ATcompare* can be found in Figure 5.10. The basic idea of the function is that first a check is made if the two terms are equal. If the terms are equal 0 is returned. If they are not, the types of the terms are compared and if the types are not equal, either -1 or 1 is returned. If the types are equal, but the terms are not equal, then depending on the type, additional code is executed. The code contains a bug. In the if statement, the first comparison between the types is the same as the second one. $a < b$ is equal to $b > a$ hence the second branch is never executed. More importantly, if the type of the first term is bigger than the type of the second term, the *switch* statement is executed and further calculations are made assuming the type of the second term is equal to the type of the first term. So when a real is compared to an integer, the switch statement will eventually be executed and further calculations will be made assuming both terms are reals. This will cause the program to crash. The second condition in the if statement has to be changed to $type2 < type1$.
Using GCOV, we got the following coverage data after executing the test file based on the 88 quantifications:

- Line/Statement Coverage: **69,30**%
- Branch Coverage: **51,50**%
- Calls executed: **49,65**%

The code that was covered by this test file has a lot of similarities with the code that was covered by the existing test file, but there was also some code which was not covered by the *stress.c* file (for instance some of the 49 functions which were not covered by that file). The *stress.c* file on the other side covered some code which was not executed by our test file. We explain why in the next section

```
int ATcompare(ATerm t1, ATerm t2)
{
  int type1;
  int type2;
  int result = 0;

  if (ATisEqual(t1, t2)) {
    return 0;
  }

  type1 = ATgetType(t1);
  type2 = ATgetType(t2);

  if (type1 < type2) {
    return -1;
  }
  else if (type2 > type1) {
    return 1;
  }

  switch (type1) {
    case AT_APPL:
      ...
    case AT_INT:
      ...
    case AT_REAL:
      ...
  }
  ...
}
```

Figure 5.10: Definition of *ATcompare*

5.5.1 Shortcomings

Compared to the *stress.c* file, every coverage metric is a bit smaller, though the difference is not that big compared to the coverage data of the original test file. Some reasons for not covering as much code as the test file and not reaching 100% code coverage are:

- **Specific test code**. Some tests in the *stress.c* file are specific for a certain set of input values and cannot be generalised and used in a quantification. Consider for instance a checksum test function in *stress.c*:

```
void testChecksum()
{
  ATerm t = ATparse("f(a,b,1,2,[])");
  char expected_digest[16] = { 0xf0, 0xbb, 0xaf, 0x3d, 0x93, 0xfa, 0x08, 0x2c,
    0xb7, 0xfe, 0xa9, 0x79, 0x6c, 0xd5, 0xdd, 0xdd };

    test_assert("checksum", 0, memcmp(expected_digest, ATchecksum(t), 16) == 0);

    printf("checksum tests ok.\n");
}
```

  This function tests *ATchecksum* by comparing the result of that function to a series of characters. This test is specific to the input value $f(a, b, 1, 2, [])$ and is difficult to generalize for all terms. Hence no quantification is present using *ATchecksum*. More knowledge about some of the functions is necessary to be able to create an adequate quantification describing their working.

- **Input values cannot be generalised**. The functions *ATmake* and *ATmatch* are an important part of the library. However the input values for these input values cannot be generalised. In both cases the first argument, which is a pattern, determines the number of (and also the types of) additional arguments for those functions. Consider the following three examples of creating an ATerm:

```
ATerm t1,t2,t3;

int i=3;
double r=5.5;
char * s1="f";
char * s2="g";

t1=ATmake(<int>,i);
t2=ATmake(<real>,r);
t3=ATmake([<appl(2),<appl(3,5,6)>],s1,s2);
```

  The first call to *ATmake* has 2 arguments of which the second is an integer. The second call has a real as second argument and the third call even has 3 arguments. Hence besides the obvious pattern containing only 1 placeholder, we did not specify *ATmake* or *ATmatch* in a quantification. The ATerm library also has some print functions which are similar to their C counterparts (*ATprintf* instead if *printf*, *ATfprintf* instead of *fprintf*, etcetera). Those functions have the same structure as the

ATmake function, they require a pattern followed by a number of arguments based on the pattern. Hence for these functions no quantifications were created either. When creating new tests, we have to find a way to generalize over the patterns.

- **Unreachable code**. The ATerm library contains a lot of code for handling error states, which is a good thing. However some of the error states are very difficult to reach. For instance branches which are taken when the term table is (almost) full, $NULL$ values are used or strings are parsed with an odd number of brackets are not reached. New tests have to be created, causing the library to handle these cases.

- **Blobs cannot be created easily**. Though blobs are considered to be an ATerm, they are handled quite differently from the other terms. When creating a blob, we have to specify the length of the blob, besides the data (when creating a list or a function application, you do not have to specify the length). Therefore when creating a blob 1 additional argument is necessary. Another difference between blobs and other ATerms is that two blobs are considered equal if the address of their data is equal, not if the data itself is equal (however when you compare two integer ATerms which have the same integer value, they are considered to be equal). When we want to create an ATerm using the *ATparse* function (which is done in the test cases), we cannot create a blob (the *ATparse* function does not allow this). So all the quantifications for terms are never tested with a blob as argument. Hence we missed some parts of the code. So we have to create a formalism which also handles blobs as ATerms.

5.6 Conclusion

The 88 quantifications we have used so far perform pretty well in terms of coverage compared to the 1500 lines of code in current test file. The 88 quantifications give a much better overview of the library and specify the working of the functions for every input value. Using UQT, we even found a bug, which was not found by the current test file. However there are some very specific cases which are difficult to specify. Some of these cases have to do with the limitations of UQT (patterns), others with the ATerm library (limited knowledge, blobs). With some additional research we expect to solve most of these problems. We also expect UQT to be very useful when dealing with libraries which are not as complex as the ATerm library and have a lot of functions which can be specified easily. We discuss this claim in the next section.

Chapter 6

Testing data structure libraries

6.1 Introduction

We expect UQT to work well for libraries which contain functions for which a formal specification can be made easily. Libraries implementing data structures (such as a linked list, a queue, a stack) are good candidates, since most of their functions work in isolation. We use UQT to test two data type C libraries: the LKlist library and the LibDS library. We discuss the results in the next sections.

6.2 LKlist library

The LKlist C library implements a double linked list that supports *void* data types. The library includes 14 functions that allow the user to create and manage the linked list. The library can be downloaded through Sourceforge [8]. The library contains the following functions:

```
LKAdvanceList
LKClearList
LKCreateList
LKCurrent
LKDeleteFromList
LKEmptyList
LKGetFromList
LKGotoInList
LKInsertAfterInList
LKInsertBeforeInList
LKLastInList
LKPutInList
LKReverseList
LKZapList
```

All these functions are located in 1 C file, so we do not have to create a source file of the library. The 11 quantifications we made for the functions can be found in Appendix E. The only domain we used in the quantifications is Int, which we used for the number of elements in the list and the actual values of the

elements. Hence the data file contained only 1 line, defining the representative values 0, 1, 5 and 200 for the Int domain. We choose these values to test some different sizes of the linked list, but we also did not want to use too many elements, since we are not able to free any memory.Executing the test file took about 2 seconds and resulted in checking 121331 assertions. None of these assertions were violated, so the library works as expected. The following coverage data was gathered using GCOV:

- Line/Statement Coverage: **95.35**%
- Branche Coverag: **77.78**%
- Calls executed: **75.00**%

The only parts of the library that were not executed were branches that were reached when a memory allocation error occured. With only 11 quantifications and 1 domain definition we were able to test all the code that we wanted and we may safely assume that the library works as expected.

6.3 LibDS library

The LibDS library [7] is a C library which implements some data structures and functions to manipulate them. The data structures that are implemented are a heap, an AVL tree (a balanced binary tree), a hashtable, an array, a queue, a set and a stack. The functions of the library can be found in Appendix F. Apart from 7 source files for every data structure, the source of the library has a folder *doc* with an HTML file for every data structure describing the functions of the data structure. However these functions do not correspond to the code, some functions, such as $avlMake$, are described in an HTML file, but not defined in the source files and some functions, like $avlNodeKey$ are defined in the code but not mentioned in an HTML file. This makes testing the library time consuming, since the expected output of some functions has to be derived purely based in the source code (which does not contain a lot of comment or explanations). The library also provides a test file for every data structure. Executing these test files for every data structure resulted in the following coverage metrics:

Data structure	**Statement Coverage**	**Branch Coverage**	**Calls executed**
Heap	50.77% of 260	45.52% of 134	50.85% of 59
AVL tree	31.97% of 610	20.09% of 458	29.77% of 131
Hashtable	72.26% of 137	48.44% of 64	70.37% of 27
Array	62.50% of 112	42.86% of 56	57.89% of 19
Queue	38.42% of 177	31.58% of 76	42.31% of 26
Set	62.88% of 132	52.17% of 92	67.16% of 67
Stack	56.52% of 46	33.33% of 12	50.00% of 10
Total	45.65% of 1474	31.84% of 892	47.20% of 339

Not even one half of the code is covered by these test files, we expect to do better using UQT. Some of the first executions of the test file caused the system to crash. After rewriting or removing the quantifications that had functions which caused the system to crash we came to a total of 64 quantifications which can be found in Appendix G. The data file contained only 2 domains: Int contains the values 1, 3, 4, 10, 55 and 200. We did not use 0 because this caused some of the functions to crash. We also had some problems with some *Close* functions, so we choose not to create more than 200 elements, to make sure that

no problems occur with memory allocation (we could not use the *Close* functions to free memory). The $SortMethod$ domain contains the values 0 and 1 (sorting a data structure either ascending or descending). The test file was executed and finished in 8 seconds. 846 856 Assertions were tested, of which 41 failed. All those assertions occured in a test case in which a $CloseWithFunction$ was used. Further investigation of the code led to the following bugs:

- **0 values**. The implementation takes 1 as the first index for an element (instead of 0). Using 0 for the size of a data structure causes some functions to crash (likely because the implementer often uses 1 as the starting point). Trying to retrieve an element at index 0 also causes the library to crash.
- **Summation of the same reals leads to different values**. The library contains several $Walk$ functions which traverse a data type. These functions take another function as argument which will be executed on every element of the data type. We used this function to add the value of the elements to a variable. When initializing the data structure we also added the value of every element to a variable. Hence we expect the two variables to be equal. When we use integer elements, this is the case, however when we use reals, the values are not equal. The difference is very little (in the order of 1 / 10000000), but we expect that not enough memory is allocated for each element.
- $Next$ **and** $Prev$ **do not work for large data structures**. Quantifications were created which traversed certain data structures using the $Next$ and $Prev$ functions. These functions work as expected for small lists, however when dealing with large data structures, they seem to skip an element. We created an array and added 200 values to it. When we traversed the array using the $paFirst$ and $paNext$ function, only 199 elements were traversed (we used an additional variable to check this). The $paFirst$ functions works as expected, we checked this in another quantification, hence either something is wrong with the storage of the elements or something is wrong with the $paNext$ funtion. Due to the complexity of the code, we were not able to locate the exact bug, but we can describe the problem.
- **Close functions crash**. Most data structures have a *Close* function and a $CloseWithFunction$ function, the difference being that the $CloseWithFunction$ also performs a function on each element of the data structure. Most $CloseWithFunction$ contains a bug: no check is made if the function which will be executed on each element is valid (at least a check has to be made if the function is not NULL). The functions *avlClose* and *htClose* even execute their respective $CloseWithFunction$ with the function $NULL$ and hence the system crashes when trying to execute these functions.
- **Wrong type in paContains**. The function *paContains* checks if an array contains a certain element. The function uses a compare function as argument and the result has to be 0 if the elements are equal and 1 or -1, depending on the type, if they are not. The code for *paContains* is:

```
int
paContains(PARRAY pa,int (*cmpfun)(void*,void*),void *elem)
{
    int i;
    PArray * parray = (PArray*) pa;

    for (i = 0; i < parray->num_elems; ++i)
    if (cmpfun(parray->elements[i],elem))
      return i;
    return -1;
```

```
}
```

The code uses the compare function as a boolean, not an integer so if the function always returns a value, the condition evaluates to $TRUE$ and returns an incorrect index. The line:

```
if (cmpfun(parray->elements[i],elem))
```

should be replaced by:

```
if (cmpfun(parray->elements[i],elem)==0)
```

The following coverage data was gathered after executing the test file:

Data structure	**Statement Coverage**	**Branch Coverage**	**Calls executed**
Heap	92.31 % of 260	73.13% of 134	94.92% of 59
AVL tree	79.84% of 610	70.31% of 458	65.65% of 131
Hashtable	91.24% of 137	81.25% of 64	85.19% of 27
Array	92.86% of 112	75.00% of 56	100.00% of 19
Queue	85.88% of 177	69.74% of 76	92.31% of 26
Set	93.18% of 132	80.43% of 92	98.51% of 67
Stack	100.00% of 46	83.33% of 12	100.00% of 10
Total	**86.64%** of 1474	**73.45%** of 892	**83.78%** of 339

We covered almost all the code except for the branches reached when a memory allocation error occured. The source code for the AVL tree also contained some functions like *avlCut* which were very complex and unclear. We could not create a quantification for those functions. Apart from that, we are very pleased with the amount of code we covered and the bugs we were able to find.

6.4 Conclusion

We used UQT to test 2 libraries of different size, but both with a satisfying result. Using only 11 and 64 quantifications respectively, we were able to test 95% and 87% of the code. In the case of the second library we were also able to find some bugs, which were not found by the existing test files. Again as with the ATerm library, the difficulty of testing and specificying code that is not well documented resulted in a time consuming process. However using the test file as a basis we were able to generate the tests quickly and efficiently. However we have encountered some problems using UQT. One of those problems is that it is not possible to see the values of the assertions that failed, the CUnit framework only displays the boolean expression. We want use a certain hierarchy for the quantifications as well, we want to group the 64 quantifications used for the LibDS library for every data structure. In the next chapter we will look at how to adapt the framework to solve those problems.

Chapter 7

Improving UQT

We were able to test the two data structure libraries quite successfully, but some adaptations have to be made to UQT to make the debugging a bit easier. We also have some problems to solve which occured when we tested the ATerm library. Taking all problems into consideration, we made some improvements to UQT and the related files which we will discuss now.

7.1 Multiple test suites

We want to be able to group certain quantifications. Therefore we allow the user to specify the test suite to which a quantification belongs. By adding the line "−−Test suite name" to the quantification line, all the following quantifications will be added to that test suite until a new test suite is specified. If the file contains the following text:

```
--Test suite 1
Quantification-1
Quantification-2
--Test suite 2
Quantification-1
Quantification-2
Quantification-3
```

The first two quantifications will be grouped and the last three quantifications will be grouped. When testing the ATerm library we can for instance group the different print functions, make functions and list functions using this construct.

7.2 Code snippets

Some quantifications result in the creation of a lot of data and hence a lot of memory usage. For instance using a quantification several copies of an array can be made. Ideally after the predicate has been checked, the memory occupied by the variable is released. At the end of the quantification we want to be able to execute certain code to "clean up the memory" or achieve similar goals by executing additional code. We

cannot achieve this using the quantifications, the code snippets are only allowed in the range of the quantification. Therefore we will allow a predicate to consist of code snippets besides boolean expression. The grammar for our quantification language contains one additional rule, rule predicates-2:

[quantification]	Quantification	→	"(∀" Declarations " : " Predicates ")"
[declarations-0]	Declarations	→	Declaration
[declarations-1]	Declarations	→	Declaration " ∧ " Declarations
[declaration-0]	Declaration	→	INT "≤" VAR "<" INT
[declaration-1]	Declaration	→	VAR " in " DOMAIN
[declaration-2]	Declaration	→	"(" TYPE ")" VAR " in " DOMAIN
[declaration-3]	Declaration	→	"[[" CODE "]]"
[predicates-0]	Predicates	→	Predicate
[predicates-1]	Predicates	→	Predicate " ∧ " Predicates
[**predicates-2**]	**Predicates**	→	**Predicates** ∧ **"[[" CODE "]]"**
[predicate-0]	Predicate	→	BOOLEAN
[predicate-1]	Predicate	→	Quantification

We want to ensure that the predicate of a quantification always contains a boolean expression, hence we added the code snippets in a rule for *Predicates*, not in the rule for *Predicate*. If we added the rule *Predicate* → "[[CODE "]]", then the predicate could contain only a code snippets and no boolean expressions. This is undesirable, since a quantification does not have any meaning without a predicate. By changing the grammar, the following quantifications are valid:

(∀ i: (Int)i ∈ ℤ ∧ [[XTable table=XMake(10);XAdd(i);]]: XSize(table)==1 ∧ [[XDestroy(table);]])

(∀ i: (Int)i ∈ ℤ ∧ [[YArray array=YMake(20); int j=YInsert(arra,i,i);]]: YSize(array)==1
∧ [[YDestroy(array);]] ∧ j==i)

7.3 Value list

It is difficult to find out why some assertions are violated and some are not. The CUnit frameworks only displays the boolean expression that evaluated to false, not the values of the variables. So if we test 20 values for a predicate and 10 cause an assertion to fail, then we are not presented with the exact values (and get better insigth into what is causing the failure). Consider for instance a quantifcation which has one bounded variable: i and the predicate is $P(i)$. If for 3 values of i, the predicate is false, CUnit only displays the boolean expression of the assertion, in this case $P(i)$. However of interest to us is the actual value of i. Therefore we keep track of the values of every bounded variable in a linked list. If a value is parsed into a bounded variable, the value together with the name of the bounded variable is added to the list. If the scope of the bounded variable ends, the element is removed from the list. We also use a modified version of the assert version we used so far (CU_ASSERT). We not only pass the boolean expression as argument, but also a string, which contains the values of the bounded variables in the boolean expression. This new function is defined as CU_ASSERT_VALUES as follows (the definition of CU_ASSERT is also given to see the similar code):

```
#define CU_ASSERT(value) \
  { CU_assertImplementation((value), __LINE__, #value, __FILE__, "", CU_FALSE); }

#define CU_ASSERT_VALUES(value,vallist) \
  {  char *zng; zng=malloc(sizeof(char*)*1000); strcpy(zng,#value); \
     strcat(zng,", with "); strcat(zng,vallist); \
     CU_assertImplementation((value), __LINE__, zng , __FILE__, "", CU_FALSE); \
     free(zng);  }
```

The third argument of the function *CU_assertImplementation* is (string representation of) the boolean expression, so we added the values of the bounded variables to that string.

The code generated by the quantification:

(∀: (Int)i ∈ Int ∧ (Int)j ∈ Int ∧: $i + j > 1$)

can be found in Figure 7.1. Every variable value pair is added to the linked list right before the value is parsed. At the end of the scope of the variable (so right before the closing bracket of the *while* loop) the last element of the linked list is removed (using the *pop* function).

Executing the code with the domain specification Int={0,1,2,3,4321,7654321} will result in a XML file which has information about the value of the bounded variables as can be seen in Figure 7.2. We can now see which of the values of the bounded variables cause the assertion $i + j > 1$ to fail: (i=0 and j=0), (i=0 and j=1) and (i=0 and j=0).

7.4 Handling blobs

A problem in the first test run was that no blobs could be parsed into an ATerm variable based on a string. Hence we added the following line right after the line which parsed a string into an ATerm (VNAME is used for the name of the variable):

```
if(strstr(VNAME_list->data,"(blob)")-(int)VNAME_list->data==0)
  VNAME=(ATerm)ATmakeBlob(strlen(VNAME_list->data),strdup(VNAME_list->data));
```

This line ensures that when the value of a domain starts with "(blob)", the variable is declared as a blob and the data used for the blob is the remaining part of the variable. So if we range over ATerms, we can define the value "(blob)aa" in the data file and a blob with value "aa" will be parsed into the bounded variable.

```
void testcase0() {
  /*  value list will contain the values for the bounded variable */
  struct node *value_list;
  value_list=NULL; append(&value_list,"start");
  /* this variable will be used to create the string "variable name = value"
  char *r;  r=malloc(sizeof(char)*1000);
  /* used to parse an integer value into a string.
     This is necessary when a variable is defined like this: 0<=i<200
     Then no linked list is used for the values */
  char *itoabuffer=malloc(sizeof(char)*1000);

  struct node *i_list;
  i_list=NULL; append(&i_list,"start");
  isDataType("Int","data.cps",i_list);
  while (i_list->link != NULL) {
    i_list=i_list->link;

    /* the value and name of a bounded variable are concattenated and added to the list */
    strcpy(r,"i = "); strcat(r,i_list->data); append(&value_list,r);

    int i; i=atoi(i_list->data);
    struct node *j_list;
    j_list=NULL; append(&j_list,"start");
    isDataType("Int","data.cps",j_list);
    while (j_list->link != NULL) {
      j_list=j_list->link;

      strcpy(r,"j = "); strcat(r,j_list->data); append(&value_list,r);
      int j; j=atoi(j_list->data);

      /* getString is used to get a string representation of the linked list */
      CU_ASSERT_ATERM(i+j>1,getString(value_list));

      /* remove the value of j */
      pop(&value_list);
    };
    /* remove the value of i */
    pop(&value_list);
  };
  free(r);
  free(itoabuffer);free(value_list);
}
```

Figure 7.1: Code including the value list

CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Running Suite global

Running test TestCase0 ...		Failed	
File Name	testoutput.c	**Line Number**	41
Condition	i+j>1, with [i = 0][j = 0]		
Running test TestCase0 ...		Failed	
File Name	testoutput.c	**Line Number**	41
Condition	i+j>1, with [i = 0][j = 1]		
Running test TestCase0 ...		Failed	
File Name	testoutput.c	**Line Number**	41
Condition	i+j>1, with [i = 1][j = 0]		

Cumulative Summary for Run

Type	Total	Run	Succeeded	Failed
Suites	1	1	- NA -	0
Test Cases	1	1	0	1
Assertions	25	25	22	3

Figure 7.2: Adapted report

7.5 Generating data / Handling patterns

When choosing representative values for the domain in the quantifications we have not used any rules, algorithms or formalizations. We only used our common sense to choose some representative values. When looking at the *Integer* domain, we choose 0 and 1 (among some other values), because those values are at the boundary of the domain. But what about ATerms? What are the "special" cases for that domain? What terms will likely cause problems and what set of values are a good representation of the domain? We cannot give an answer on those questions using only our common sense. However we can give a context free grammar (see section 4.3.4 for more information about context free grammars) of the ATerm language (the language containing all valid ATerms). We can then use a form of grammar coverage to gain confidence in the fact that we have a representative set of values for the language. We first present some coverage criterions for grammars and after giving a grammar for the ATerm language, we will apply the most suited coverage criterion to the ATerm language to generate a (test) set of ATerms.

7.5.1 Grammar coverage

Most grammars can produce an infinite number of words. So we cannot test all the words a grammar can produce but only a representative set of words. Several coverage criterions exist for grammars. The most used form of grammar coverage is rule coverage, first introduced by Purdom [32]. Rule coverage simply means that a test set explores all rules of a grammar. Lämmel and Schulte introduce the notion of controllable combinatorial coverage [25], which is an approximation of complete combinatorial coverage (combinatorial coverage is reached when every word that can possibly be generated by the language is used). Controllable combinatorial coverage uses some restrictions, like the number of rules that are used, to limit the number of words. Lämmel also introduced the notion of context-dependent rule coverage, a generalisation of rule coverage such that all the different occurrences of a nonterminal are distinguished [24]. Another coverage criterion is two-dimensional approximation coverage, which uses recursive depth in the definition of coverage

[3]. We will use context-dependent rule coverage as the coverage criterion for the ATerm language because it is a stronger coverage notion than rule coverage [24]. We also expect that achieving controllable combinatorial coverage will still require a lot of terms. Creating a list with 3 elements for every possible ATerm subtype requires 6^3=216 terms, not even taking annotations into account. Since we will use context-dependent rule coverage we will give some definitions.

Definition 1. *Let $G = \langle N,T,s,P\rangle$ be a context-free grammar. If $m \rightarrow u\ n\ v \in P$, where $m,n \in N$, $u,v \in (N \cup T)$, then $m \rightarrow u\ n\ v$ is called direct occurrence of n in G. Occs(G,n) denotes the set of all direct occurrences of n in G.*

Example. Consider the following grammar, grammar G1, in which the start symbol is s:

[r1]	s	$\rightarrow$	A B
[r2]	A	$\rightarrow$	B a
[r3]	B	$\rightarrow$	ϵ
[r4]	B	$\rightarrow$	b B

This grammar leads to the following direct occurences:

Occs(G1,s)	$= \emptyset$
Occs(G1,A)	$=$ {[r1] s $\rightarrow$ A B}
Occs(G1,B)	$=$ {[r1] s $\rightarrow$ A B, [r2] A $\rightarrow$ B a,[r4] B $\rightarrow$ b B}

Definition 2. *Let $G = \langle N,T,s,P\rangle$ be a context-free grammar. $w \in L(G)$ is said to cover the rule $p = n \rightarrow z \in P$ for the occurrence $m \rightarrow u\ n\ v$ if there is a derivation $s \rightarrow^*_G x\ m\ y \rightarrow^q_G x\ u\ n\ v\ y \rightarrow^p_G x\ u\ z\ v\ y \rightarrow^*_G w$. $W \subseteq L(G)$ is set to cover $p = n \rightarrow z \in P$ for all occurrences, if there is a $w \in W$ for all occurrences $o \in Occs(G,n)$ such that w covers p for o. W is said to achieve context-dependent rule coverage (CDRC) for G, if W covers all $p \in P$ for all occurrences.*

Example. Consider the following grammars, R and T (for both grammars the start symbol is s):

R:				**T:**			
[r1]	s	$\rightarrow$	A B	[t1]	s	$\rightarrow$	A B
[r2]	A	$\rightarrow$	B a	[t2]	A	$\rightarrow$	a
[r3]	B	$\rightarrow$	ϵ	[t3]	B	$\rightarrow$	ϵ
[r4]	B	$\rightarrow$	b B	[t4]	B	$\rightarrow$	b B

The test set $\{ab\}$ achieves rule coverage for both R and T. However since r2 $\in$ Occ(R,B) we have to create a word using r2 to achieve CDRC for R. The test set $\{a, babb\}$ achieves CDRC for R. The test set also seperates R from T, since $babb \in$ L(R) and $babb \notin$ L(T). This example illustrates why CDRC is stronger than simple rule coverage.

We use CDRC as the coverage criterion for the ATerm grammar which we will present in the next section. We construct a set of representative values such that set achieves context-dependent rule coverage for the ATerm grammar.

[start]	Start	→	ATerm
[aterm-0]	ATerm	→	Int
[aterm-1]	ATerm	→	Real
[aterm-2]	ATerm	→	Appl
[aterm-3]	ATerm	→	List
[aterm-4]	ATerm	→	Placeholder
[aterm-5]	ATerm	→	Blob
[aterm-6]	ATerm	→	ATerm "{" Annotations "}"
[appl-0]	Appl	→	Symbol "()"
[appl-1]	Appl	→	Symbol "(" FArguments ")"
[list-0]	List	→	"[]"
[list-1]	List	→	"[" LArguments "]"
[farguments-0]	FArguments	→	ATerm
[farguments-1]	FArguments	→	ATerm "," FArguments
[larguments-0]	LArguments	→	ATerm
[larguments-1]	LArguments	→	ATerm "," LArguments
[annotations-0]	Annotations	→	Annotation
[annotations-1]	Annotations	→	Annotation "," Annotations
[annotation-0]	Annotation	→	"[" ATerm "," ATerm "]"

Figure 7.3: GATerm, grammar for the ATerm language

7.5.2 ATerm grammar

The grammar for the ATerm language is a quadruple $\langle N, T, s, P \rangle$. The set of non terminals N contains s, *ATerm*, *Int*, *Real*, *Appl*, *List*, *Placeholder*, *Blob*, *Annotations*, *Annotation*, *Symbol* and *Arguments*. For the *Int*, *Real*, *Blob* and *Symbol* non terminals we use boundary values together with some random values. For instance we will use the values 0, 1, 2, 10, 123456, 200000, and 4294967295 to create an ATerm using the *Int* nonterminal. We will discuss how we have chosen the values for the *Placeholder* domain in the next section. Figure 7.3 contains the set of production rules.

Note that the syntax for arguments of a function application is the same as the syntax for the arguments of list. However we expect the library to handle those arguments differently and we want to represent this in the grammar. Hence we added 2 non terminals, *FArguments* and *LArguments* for the arguments of function applications and lists instead of 1 non terminal.

We want to define a set of ATerms such that context-dependent rule coverage is achieved for this grammar. We start with calculating the set of occurences of every non terminal. Some examples from the ATerm grammar are:

Occs(GATerm,Annotation)	= {	[annotations-0] Annotations → Annotation, [annotations-1] Annotations → Annotation "," Annotations}
Occs(GATerm,FArguments)	= {	[appl-1] Appl → Symbol "(" FArguments ")", [farguments-1] FArguments → ATerm "," FArguments}
Occs(GATerm,ATerm)	= {	[start] Start → ATerm, [aterm-6] ATerm → ATerm "{" Annotations "}", [farguments-0] FArguments → ATerm, [farguments-1] FArguments → ATerm "," FArguments, [larguments-0] LArguments → ATerm, [larguments-1] LArguments → ATerm "," LArguments, [annotation-0] Annotation → "[" ATerm "," ATerm "]" *(first occurence)*, [annotation-0] Annotation → "[" ATerm "," ATerm "]" *(second occurence)*}

The complete set of occurences for every non terminal can be found in Appendix H. To achieve CDCR, every rule in the occurence of a non terminal has to be followed by every rule with the non terminal on the left side of a production rule. Consider for instance rule *aterm*-0, which "creates" an integer from an ATerm. For every rule in Occs(GATerm,ATerm) we have to use *aterm*-0 for the ATerm non terminal on the right side of that rule. This means that to cover the rule *aterm*-0 we have to create the following ATerms:

- An integer ATerm (integer being the main type, so the term does not occur in a list or function application (*start*).
- An integer ATerm with annotation (*aterm*-6).
- An integer ATerm which is the last element of a function application ($farguments$-0).
- An integer ATerm which is an argument of a function application, but not the last one ($farguments$-1).
- An integer ATerm which is the last element of a list ($larguments$-0)
- An integer ATerm which is in a list, but not the last element ($larguments$-1)
- An integer ATerm which is used as the key of an annotation (*annotation*-0)
- An integer ATerm which is used as the value of an annotation (*annotation*-0)

Besides handling all these cases for integer ATerms, we also have to handle them for reals, placeholders and the other ATerm subtypes. Though this might seem to lead to a lot of ATerms, certain cases can be combined. For instance the list $[2, 3.5]$ uses rule [larguments-1] for an integer ATerm and rule [larguments-0] for a real ATerm. Taking all these restrictions into account we created a set of ATerms manually such that CDCR was achieved for the ATerm grammar (we did not use a program since we expect that it takes quite some time to create a program which can combine different cases). We constructed this set in a way such that CDRC is achieved by construction. By ensuring that every production rule is used for every occurence, CDRC is achieved. For the ATerm language this impies using every subtype as the last and not the last argument of a list and function application. We also did not create a list or function application with a depth bigger than 2 (so a list inside a list is allowed, but a list, inside a list, inside another list is not allowed). We did this because we feel that a list containing a function application which has an integer argument is handled the same as a list containing a function application with a real argument. However we expect a list

with an integer to be handled differently from a list with a real. Without giving values for the non terminals *Int*, *Real*, *Placeholder*, *Sym* or *Blob*, the terms in Figure 7.4 achieve CDRC for the ATerm Grammar. Note that we have used *Sym*(..) for a function application with arbitrary arguments. To achieve context dependent rule coverage we only have to use rule [*aterm*-2] to create a function application in those cases, there are no restrictions on the type and the numbers of arguments. When we want to use these terms for testing purposes, these blanks have to be filled in (but the arguments are not relevant for the coverage of the grammar).

The words in Figure 7.4 are not ATerms, since the values for the *Int* non terminal are not used. Assume we use the set {0,1,2,3,4321,7654321} for the *Int* not terminal, then the grammar has to be extended with the following rules:

[int-0]	Int	→	0
[int-1]	Int	→	1
[int-2]	Int	→	2
[int-3]	Int	→	3
[int-4]	Int	→	4321
[int-5]	Int	→	7654321

Since Occs(GATerm,Int)= {[aterm−0] ATerm → Int }, to achieve CDRC, we have to use these values only once in the terms in Figure 7.4, since there are 23 occurences of *Int*, we have enough places to use the values. The same holds for the *Real*, *Sym* and *Blob* when we use no more than 17 values.

Placeholders

Choosing representative values for the *Placeholder* non termninal is not straightforward, since the definition of a placeholder is recursive. However since we can give a grammar for the language of placeholders, we can again apply the principle of CDRC to that grammar to get a representative set of placeholders. The grammar for the placeholder language (GPlaceholder):

[start]	Start	→	Placeholder
[placeholder−0]	Placeholder	→	"<int>"
[placeholder−1]	Placeholder	→	"<real>"
[placeholder−2]	Placeholder	→	"<placeholder>"
[placeholder−3]	Placeholder	→	"<list>"
[placeholder−4]	Placeholder	→	"<blob>"
[placeholder−5]	Placeholder	→	"<term>"
[placeholder−6]	Placeholder	→	"<str(" SArgs ")>"
[placeholder−7]	Placeholder	→	"<appl(" FArgs ")>"
[placeholder−8]	Placeholder	→	"[" LArgs "]"
[sargs-0]	SArgs	→	Placeholder
[sargs-1]	SArgs	→	Placeholder "," SArgs
[fargs-0]	FArgs	→	Placeholder
[fargs-1]	FArgs	→	Placeholder "," FArgs
[largs-0]	LArgs	→	Placeholder
[largs-1]	LArgs	→	Placeholder "," LArgs

Int
Int $\{[Int, Real]\}$
$Real$
$Real$ $\{[Sym(..), Int], [Real, Placeholder], [List, Blob]\}$
$Blob$
$Blob$ $\{[Real, Placeholder], [Int, Sym(..)\}$
$Placeholder$
$Placeholder$ $\{[List, Blob], [Int, Real], [Placeholder, List]\}$
$Sym(Real, List)$ $\{[Blob, Sym(..)], [Sym(..), Sym(..)]\}$
$Sym(Blob, Placeholder, Sym())$
$Sym(Int, Sym(), Blob)$ $\{[Real, Placeholder]\}$
$Sym(List, Int)$
$Sym(Real, Sym())$ $\{[Placeholder, List], [Blob, Sym(..)]\}$
$Sym(Int, Int, Placeholder)$
$Sym(Sym(), Sym(), Real)$ $\{[Int, Real]\}$
$Sym(Int)$
$Sym(Real)$ $\{[Placeholder, List], [Blob, Sym(..)]\}$
$Sym(Blob)$
$Sym(Placeholder)$ $\{[Sym(..), Sym(..)], [Sym(..), Int]\}$
$Sym(Sym())$
$Sym(List)$ $\{[List, List]\}$
$[Real, List]$
$[Blob, Placeholder, Sym()]$
$[Int, Sym(), Blob]$ $\{[Int, List], [Blob, Placeholder]\}$
$[List, Int]$
$[Real, Sym()]$ $\{[Real, Int], [Int, Blob]\}$
$[Int, Int, Placeholder]$ $\{[Sym(), Sym()], [Sym(), Int]\}$
$[Sym(), Sym(), Real]$ $\{[List, List]\}$
$[Int]$
$[Real]$ $\{[Placeholder, Int], [Blob, Real]\}$
$[Blob]$
$[Placeholder]$ $\{[Blob, Real]\}$
$[Sym()]$
$[List]$ $\{[Real, Sym(..)]\}$

Figure 7.4: ATerms that achieve CDCR

Note that we again have used 3 non terminals instead of 1 for the arguments of a quoted function application (<str>), non quoted application (<appl>) and a list ([ ]). This leads to the following set of occurrences:

Occs(GPlaceholder,SArgs)	= {	[placeholder-6] Placeholder → "<str(" SArgs ")>", [sargs-1] SArgs → Placeholder "," Sargs }
Occs(GPlaceholder,FArgs)	= {	[placeholder-7] Placeholder → "<appl(" FArgs ")>", [fargs-1] FArgs → Placeholder "," Fargs }
Occs(GPlaceholder,LArgs)	= {	[placeholder-8] Placeholder → "[" FArgs "]", [largs-1] LArgs → Placeholder "," Largs }
Occs(GPlaceholder,Placeholder)	= {	[start] Start → Placeholder, [sarguments-0] SArguments → Placeholder, [sarguments-1] SArguments → Placeholder "," SArguments, [farguments-0] FArguments → Placeholder, [farguments-1] FArguments → Placeholder "," FArguments, [larguments-0] LArguments → Placeholder, [larguments-1] LArguments → Placeholder "," LArguments }

We again created a test set manually. The placeholders in Figure 7.5 achieve context dependent rule coverage for the placeholder grammar. The representative set contains 48 placeholders, so 48 rules have to be added to the ATerm grammar for creating the placeholders. The set of terms in Figure 7.4 contains only 16 references to placeholders (the *Placeholder* non terminal occurs only 16 times). Hence we have to create an additional term, which is a list of all the placeholders in Figure 7.5. By doing this CDRC is achieved for the ATerm grammar (since *Placeholder* only occurcs in the rule $ATerm \rightarrow Placeholder$, the only obligation we have is to create all the placeholders). Combining all this information leads to the data file that can be found in Appendix I.

Patterns

Note that when we first tested the ATerm library, we had some problems with functions which use placeholders (patterns), such as *ATmake* and *ATmatch*. The functions require different types of arguments and a different number of arguments depending on the pattern, which is the first argument of the function. Since we have created a set of 48 placeholders which cover the placeholder grammar. We will use these placeholders to create an equal amount of quantifications specifying the functions *ATmake* and *ATmatch*. For instance 2 patterns are "<str(<term>)>" and "[<int>]". For these patterns we create the following quantifications:

(∀: (String)s ∈ QSymbol ∧ t ∈ Term ∧ [[ATerm aterm=ATmake("<str(<term>)>",s,t);
char *res0;ATerm *res1;
ATmatch(aterm,"<str(<term>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t))

(∀: (Int)i ∈ Int∧ [[ATerm aterm=ATmake("[<int>]",i);
int *res0;
ATmatch(aterm,"[<int>]",&res0);]]:
res0==1)

<int>
<real>
<placeholder>
<list>
<blob>
<term>
<appl(<int>)>
<appl(<real>)>
<appl(<placeholder>)>
<appl(<list>)>
<appl(<blob>)>
<appl(<term>)>
<str(<int>)>
<str(<real>)>
<str(<placeholder>)>
<str(<list>)>
<str(<blob>)>
<str(<term>)>
[<int>]
[<real>]
[<placeholder>]
[<list>]
[<blob>]
[<term>]
<appl(<real>,<placeholder>)>
<appl(<list>,<blob>,<str()>,<real>)>
<appl(<placeholder>,<appl()>,<list>)>
<appl(<blob>,<appl()>)>
<appl(<int>,<int>,<str()>)>
<appl(<term>,<blob>,<blob>)>
<appl(<list>,<list>,<int>)>
<appl(<term>,<term>,<term>)>
<str(<real>,<placeholder>)>
<str(<list>,<blob>,<str()>,<real>)>
<str(<placeholder>,<appl()>,<list>)>
<str(<blob>,<appl()>)>
<str(<int>,<int>,<str()>)>
<str(<term>,<blob>,<blob>)>
<str(<list>,<list>,<int>)>
<str(<term>,<term>,<term>)>
[<real>,<placeholder>]
[<list>,<blob>,<str()>,<real>]
[<placeholder>,<appl()>,<list>]
[<blob>,<appl()>]
[<int>,<int>,<str()>]
[<term>,<blob>,<blob>]
[<term>,<term>,<term>]
[<list>,<list>,<int>]

Figure 7.5: Placeholders that achieve CDCR

Chapter 8

Improved testing of the ATerm library

In the previous chapter we explained how we adapted UQT and created a new data file. Apart from that we also took a closer look at the code of the ATerm library to get more insight in the intended result of some functions we were not able to specify the first time. Together with the 48 quantifications for the patterns, we created 80 additional quantifications. Combined with the 88 quantifications we already had, we have a total of 216 quantifications describing the functions of the ATerm library. The complete set of quantifications can be found in Appendix J. We did not use command line options when executing the test file the first time, but we used those options to create a very small hashtable for the terms. By doing this, we ensure that the table will often be resized in different scenario's. Using UQT, the command line options and the data file from Appendix G we executed the test file which resulted in testing 216 test cases and 38 037 248 assertions. The test file took 12 minutes and 30 seconds to execute. Considering the amount of assertions and tests, this is quite fast. Of the assertions, 1763 failed which belonged to 20 test cases. Some of the quantifications ranging over terms failed when a blob was used. The reason for that was that we used an additional line to ensure that the value "(blob)a" resulted in a blob with data "a". Comparing for instance the original string ("(blob)a") to the string representation of the ATerm ("a") then resulted in a mismatch. Other quantifications failed because of bugs in the library. We will discuss these bugs in the next section.

8.1 Bugs

Further investigation of the code resulted in the following list of bugs:

- **ATwriteToString does not flush the buffer**. We created some quantifications which write terms to a file and then read that file. To write the terms to a file we used both the $ATfprintf$ function and the $fprintf$ function in combination with $ATwriteToString$. We expect the files to contain the same values, however this is not the case whenever 2 (or more) different terms are written to a file. The $ATfprintf$ function works as expected, but the $ATwriteToString$ function (in combination with the $fprintf$ does not. In the latter case the string representation of the first ATerm is used for every string which is generated using $ATwriteToString$. When this code is executed

```
ATerm t1=ATmake("<int>",1);
ATerm t2=ATmake("<real>",2.5);

FILE *file;
file = fopen("file.txt","w");

fprintf(file,"term 1 is: %s, term 2 is %s",ATwriteToString(t1),ATwriteToString(t2));
```

the file *file.txt* contains the string "term 1 is: 1, term 2 is 1" instead of "term 1 is: 1, term 2 is 2.5". We expect that some kind of bug is present in the *ATwriteToString* function which has to do with not flushing the result, but we could not find the actual error in the code.

- **Missing brackets**. The ATerm library has built in print functions which have additional features compared to the C print functions. For instance a pattern may contain "%l" to indicate that an ATermList will be written at that place. However the print functions in the ATerm library do not use the string representation we expect for a list: the brackets are missing and the comma's are not printed. So executing the following code

```
ATermList l1=ATparse("[1,2,3]");

ATprintf("%l",l1);
```

results in the string "123" being printed to the screen (and not "[1,2,3]").

- **ATdiff does not distinguish annotations**. The function *ATdiff* is used to create a pattern to indicate the difference between 2 ATerms. For instance comparing "1" and "1" results in the pattern "1", comparing "1" and "1.1" leads to "<diff-types>" and comparing "[1,2,3]" and "[1]" leads to the pattern "[1,<diff-lists>]". In some cases the *ATdiff* distinguishes 2 ATerms with equal values, but different annotations. In other cases this distinction is not made, which is inconsistent. The manual does not contain information about the expected outcome of 2 ATerms which only differ in annotation, but the inconsistency is definitively a bug: either annotations are taken into account or they are not.

- **Placeholders are always equal**. The ATerm library contains several functions to check if 2 ATerms are equal, for instance *ATisEqualModuloAnnotations*, *ATisEqual* and *AT_isDeepEqual*. Depending on different settings (for instance the type of sharing) different functions are used. However all these functions contain the same bug: 2 placeholders are always considered equal. The function *AT_isDeepEqual* contains the following code:

```
ATbool AT_isDeepEqual(ATerm t1, ATerm t2)
{
  header_type type;
  ATbool result = ATtrue;

  (..)

  type = ATgetType(t1);
  if (type != ATgetType(t2)) {
```

```
    return ATfalse;
  }

  switch(type) {
    case AT_APPL:
      { .. }
    case AT_LIST:
      { .. }
    case AT_INT:
      { .. }
    case AT_REAL:
      { .. }
    case AT_BLOB:
      { .. }
    case AT_PLACEHOLDER:
      result = AT_isDeepEqual(ATgetPlaceholder((ATermPlaceholder)t1),
      ATgetPlaceholder((ATermPlaceholder)t1));
      break;

   (..)

  }
}
```

If the two ATerms are placeholders, then the first term is compared to itself. Unfortunately, this bug occurs in all the equality functions. Clearly, the second occurence of $t1$ has to be changed to $t2$.

- **The symbol table has to initiated with a value bigger than 8**. Command line options can be used to initialize the size of different hashtables. Among those options is the option -*at-afuntable*, which initialize the symbol hashtable (a hashtable containing all the symbols used in function applications). The initial size is set to 2 to the power of the number following -*at-afuntable*, so using -*at-afuntable* 5 initializes the symbol hashtable to 2^5=32 entries. The ATerm library has built in functionality to resize a hashtable if necessary, but when we initialize the symbol hashtable with a size smaller than 2^3, the test file crashes. Code inspection did not give further insights into the reason for the crash, but apparently a small value causes problems. Note that the term hashtable can be initiated with an arbitrary (integer) value.

- **List patterns cannot be used in ATmatch if they are not the last argument**. The function *ATmatch* uses an ATerm and a pattern to store the subterms of that ATerm into additional variables. One of the types that can be used is "<list>", which represents an ATermList subterm. This list pattern can also be used as the last subterm to store several (at least 1, but possibly more) subterms in one ATermList. For instance the code

```
ATerm t1=ATmake("<int>",1);
ATerm t2=ATmake("<int>",2);
ATerm t3=ATmake("<int>",3);
```

```
ATerm term1=ATmake("[<term>,<term>,<term>]",t1,t2,t3);

ATerm t4;
ATermList list;
ATmatch(term1,"[<term>,<list>]",&t4,$list);
```

results in variable *list* having the value [2,3]. This double usage of "<list>" (both an ATermList and a variable number of arguments list) results in some problems. The "<list>" pattern can be used freely when using *ATmake*, but can only be used in *ATmatch* if it is the last pattern. Therefore the following code crashes:

```
ATerm t1=ATmake("<int>",1);
ATermList t2=ATmake("[<int>]",2);
ATerm t3=ATmake("<int>",3);

ATerm term1=ATmake("[<term>,<list>,<term>]",t1,t2,t3);

ATerm t4,t5;
ATermList list;
ATmatch(term1,"[<term>,<list>,<term>]",&t4,list,&t5);
```

since the list pattern is not used as the last one. It is very unintuitive that you can make an ATerm with a pattern, but you cannot use that same pattern to store subterms in variables. The manual also is very unclear about the usage of <list> in the patterns.

8.2 Coverage

Executing the test file resulted in the following coverage data:

File	**Statement Coverage**	**Branch Coverage**	**Calls executed**
tafio.c	85.38% of 465	62.41% of 282	82.14% of 168
memory.c	95.90% of 1049	68.96% of 973	45.00% of 280
md5.c	99.18% of 122	93.75% of 16	100.00% of 11
make.c	88.76% of 338	72.25% of 209	87.21% of 86
list.c	94.03% of 318	80.93% of 194	65.08% of 63
hash.c	82.31% of 277	59.49% of 158	67.11% of 76
gc.c	94.29% of 438	80.56% of 288	63.83% of 94
byteio.c	85.33% of 75	60.00% of 30	53.33% of 15
bafio.c	8026% of 699	65.61% of 410	73.40% of 203
gc.c	94.29% of 438	80.56% of 288	63.83% of 94
aterm.c	88.31% of 1711	73.63% of 1168	77.89% of 493
afun.c	88.09% of 235	70.90% of 134	73.08% of 78
Total	89.17% of 5727	70.90% of 3862	70.20% of 1567

No significant parts of the code were not executed. The biggest reason that we did not reach full coverage is that the library contains a lot of error handling code and code which is executed when memory allocation errors occur. We were also very pleased to see that the values we used for ATerms and patterns are a good representation of the domain. There were no branches that were not executed because a certain type of ATerm was not used. Considering the complex domain of ATerms (6 subtypes, annotations, lists and functions), we only had to use 35 terms to test the functions properly.

Chapter 9

Results and conclusions

9.1 Results

This master's project resulted in several products. Firstly, this thesis desribes an evaluation of the test suite of the ATerm library. The thesis also describes a framework, UQT, which can be used for specifying functions and generating test cases. More specifically, this thesis provides an explanation of how a representative set of values can be chosen for a domain using a grammar and the notion of context dependent rule coverage. The project also resulted in test results for 3 C libraries.

Using the GCOV tool and a manually created source file of the ATerm library, we were able to gather coverage data of the test suite of the ATerm library. The current test suite covers about 75% of the library source code and fails to execute 49 functions of the library.

To improve the testing process of the ATerm library we had to create a testing tool ourselves. UQT uses an extended version of universal quantifications to specify the results of functions. In this extended version, code snippets and type casts can be used. UQT generates a test file based on the quantifications and used some additional files and libraries to gather information about the succes of the test cases and the ammount of code that is covered by the test cases. When testing the ATerm library, we used context dependent rule coverage to determine a representative set of ATerms which we used for all the tests. The same coverage criterion is used for creating patterns which we used to test several functions.

UQT was used to test several libraries. For both the LKList and the LibDS library only a limited number of quantifications were needed to cover the code libraries as much as we wanted. We also found several bugs in the LibDS library which were not found by the test files that were presented with the library. Testing the ATerm library took some more time, because of the complexity of the library. However we were able to cover a lot of code and find some bugs, which were not found by the current test file.

9.2 Conclusions

The worked done in this thesis was aimed at improving the overall test process for the ATerm library. Early attempts showed that trying to test a library that has limited documentation is very hard. The derivation of the expected outcome of a function turned out to be not as straightforward as expected, even with the help

of a (outdated) manual. The test file presented with the ATerm library contains a lot of tests, but is not a very structered test suite. However the file gave a lot of good examples on how to use the functions and still covers about 75 % of the library source code. Unfortunately the library source file did not contain a lot of comment and the manual is not consistent with the code. A more structured approach was very desirable.

UQT provided just that. The relatively limited number of quantifications (a little over 200 to cover 16000 lines of code) covered all the parts of the library we wanted to test. Moreover the quantifications are a very compact formalism to specify the functions of the library. The quantifications model the use of a function not for a specific input value, but for an entire domain. Using UQT also a number of bugs were found that were not found by the original test file. The current set of quantification can be reused easily to test whether newer versions of the library still work as expected.

Using UQT to test other libraries proved that the framework was not ATerm specific, but could be used to test an arbitrary C library. Again only a limited amount of quantifications were necessary to cover almost the complete library and finding previously uncovered bugs.

Context dependent rule coverage proved to be an useful coverage criterion when selecting a representative set of values from a infinite domain. Both the code for the terms and the patterns contained no branches which were not executed, even though relative few terms and patterns were used.

9.3 Recommendations and future work

- **Improve error handling**. UQT does not provide very helpful error messages. If the quantifications are correct, the code is generated and the test file can be executed. However UQT does not produce a test file if there is an error in the quantification file. No error message is generated as well, so the user does not know what is wrong.
- **Improve values list**. UQT used a linked list to store all the values of the bounded variables. A stack is are more suited data structure, since the first element that is removed, is the one that is added the last. The linked list we used has to be traversed to the end everytime to add or delete a value, clearly this is not optimal.
- **Investigate domain limitations**. Currently UQT does not limit the amount of bounded variables that can be used. Since the amount tests grows exponentially with every variable, it might be wise to check the amount tests that are generated by a quantification. Together with the data in the data file, the number of tests can be calculated and perhaps even presented to the user to warn him if a quantification takes a long time to test.
- **Automatic data generation**. The representative values we have chosen for the terms and for the placeholders were created manually. It would be nice to have a tool which based on a grammar creates a set of words which achieves context dependent rule coverage for that grammar.
- **Test other versions of the ATerm library**. UQT is used to test the latest version of the ATerm library, however it would be nice to test other versions of the library as well. It would be interesting to see if previous bugs are also found by UQT.

- **Further investigate covered code**. The GCOV tool creates a coverage file for every source file that is used. These files can be used to create for instance a list of all the functions that were not covered, branches that were not taken or functions that were not completely tested. The coverage files might even be used to compare the functions used in the manual of the ATerm library to the ones actually present in the source file.
- **Test the Java version of the ATerm library**. UQT tested the C version of the ATerm library. It would be interesting to test the Java version of the library and see if an equal amount of code is covered. UQT has to be changed so that JUnit (the Java version of CUnit) code is produced.
- **Improve performance tests**. Though UQT created a lot of terms in a lot of different scenario's, no specific performance tests were used. For instance it would be interesting to see what the maximal number of elements of a list is, before the library crashes.

Bibliography

[1] Caesar/Aldebaran Development Package: http://www.inrialpes.fr/vasy/cadp.html.

[2] CUnit Unit Testing Framework: http://cunit.sourceforge.net/.

[3] Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/asml/, 2001.

[4] GCC compiler: http://gcc.gnu.org/.

[5] GCOV Coverage analysis: http://gcc.gnu.org/onlinedocs/gcc/gcov.html.

[6] ISO/IS 9074, Estelle: A Formal Description Technique Based on an Extended State Transition Model, 1989.

[7] LibDS, a Generic Data Structures Library: http://sourceforge.net/projects/libds/.

[8] LKList, a C library for handling linked lists: http://sourceforge.net/projects/lklist/.

[9] SPIN and PROMELA: http://spinroot.com/spin/whatispin.html.

[10] TorX Architecture for Test Derivation and Execution: http://www.inrialpes.fr/vasy/cadp/software/99-b-torx.html.

[11] *Stratego: A language for program transformation based on rewriting strategies*, volume 2051 of *Rewriting Techniques and Applications*. Springer-Verlag, 2001.

[12] J.A. Bergstra and P. Klint. The discrete time toolbus - a software coordination architecture. *Sci. Comput. Program*, (31):205–229, 1998.

[13] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. *Lecture Notes in Computer Science*, 2102:250–254, 2001.

[14] B. Boehm. Software and its impact: A quantitative assessment. *Datamation*, 1973.

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

[16] L. Bromstrup. Tesdl: Experience with generating test cases from sdl specification. *41th SDL Forum*, 1989.

[17] D. Chappel. *Understanding ActiveX(TM) and OLE*. MicroSoft Press, 1996.

[18] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *The Journal of Logic and Algebraic Programming*, (59):35–61, 2004.

[19] S.H. Edwards. A framework for practical, automated black-box testing of component-based software. *Softw. Test., Verif. Reliab.*, 11(2):97–111, 2001.

[20] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, 1988.

[21] J. Gosling, B.Joy, and G.Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[22] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 112–122, New York, NY, USA, 2002. ACM.

[23] Object Management Group. Omg. the common object request broker: Architecture and specification, revision 2,0. Technical report, OMG, 1997.

[24] R. Lämmel. Grammar Testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201–216. Springer-Verlag, 2001.

[25] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *TestCom*, pages 19–38, 2006.

[26] D.Y. Lee and J.Y. Lee. A well-defined estelle specification for the automatic test generation. *IEEE Transactions on Computers*, 40(4):526–542, 1991.

[27] S. Loveland, G. Miller, R. Prewitt, and M. Shannon. *Software Testing Techniques: Finding the Defects that Matter*. Charles River Media, 2004.

[28] P.-E. Moreau and O. Zendra. Gc2: a generational conservative garbage collector for the ATerm library. *The Journal of Logic and Algebraic Programming*, (59):5–34, 2004.

[29] G. Myers. *The art of software testing*. Wiley, 1979.

[30] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

[31] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, April 2007.

[32] P.W. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, (3):366–375, 1972.

[33] T.A. Sudkamp, editor. *Languages and machines : an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[34] M.G.J van den Brand. Guest editor's introduction: Special issue on Annotated Terms (ATerms). *The Journal of Logic and Algebraic Programming*, (59):1–4, 2004.

[35] M.G.J. van den Brand and P. Klint. Efficient annotaded terms. *Software Practice and Experience*, (30):259–291, 2000.

[36] M.G.J. van den Brand and P. Klint. Aterms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, (49):55–64, 2007.

[37] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, J. Scheerder P.A. Olivier, E. Visser J.J. Vinju, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[38] P.H.J. van Eijk, C.A. Vissers, and M. Diaz. *The formal description technique LOTOS*. Elsevier Science Publishers B.V., 1989.

[39] Stephen Edwards Virginia. RESOLVE: A framework for practical, automated black-box testing of component-based software.

Appendix A

Test functions in stress.c

```
testAlloc
testSymbol
testAppl
testList
testBlob
testOther
testRead
testDict
testPrintf
testAnno
testMake
testTuple
testMatch
testBaffle
testTaf
testGC
testProtect
testMark
testTable
testIndexedSet
testDictToC
testTBLegacy
testChecksum
testDiff
testCompare
```

Appendix B

Sample source code

```
#include "CUnit/Headers/Basic.h"
#include "CUnit/Headers/Console.h"
#include "CUnit/Headers/Automated.h"
#include "CUnit/Headers/CUCurses.h"
#include "library_source.c"

int init_suite_success(void) { return 0; }
int clean_suite_success(void) { return 0; }

/*

  Here the functions for every quantification are inserted

*/

int main(int argc, char *argv[]) {
  /* Initialization of the ATerm Library */
  ATerm bottomOfStack;
  ATinit(argc, argv, &bottomOfStack);
  CU_pSuite pSuite = NULL;

  /* Initialization of the CUnit test registry */
  if (CUE_SUCCESS != CU_initialize_registry())
    return CU_get_error();

  /* Add one suite to the registry for all the quantifications */
  pSuite = CU_add_suite("Test results", init_suite_success, clean_suite_success);
  if (NULL == pSuite) {
    CU_cleanup_registry();
    return CU_get_error();
  }
```

```
/*

  For every quantification the following code is executed
  X is equal to the sequence number of the test case

*/

  if (NULL == CU_add_test(pSuite, "TestCaseX", testcaseX)) {
    CU_cleanup_registry();
    return CU_get_error();
  }

/*

  The following code runs all the tests

*/

  /* Run all tests using the basic interface */
  CU_basic_set_mode(CU_BRM_VERBOSE);
  CU_basic_show_failures(CU_get_failure_list());
  /* Run all tests using the automated interface */
  CU_automated_run_tests();
  CU_list_tests_to_file();

  /* Clean up registry and return */
  CU_cleanup_registry();
  return CU_get_error();
}
```

Appendix C

Initital set of quantifications

We used the following quantifications in the first test run:

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term: !ATisEqual(t1,ATsetAnnotation(t1,t2,t3)))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term: ATisEqual(t1,ATremoveAnnotation(ATsetAnnotation(t1,t2,t3),t2)))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧ [[ATerm term1,term2; term1=ATsetAnnotation(t1,t2,t3); term1=ATsetAnnotation(term1,t2,t4);]]: ATisEqual(ATgetAnnotation(term1,t2),t4))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧ [[ATerm term1,term2; term1=ATsetAnnotation(t1,t2,t3); term2=ATsetAnnotation(t1,t2,t4); term1=ATsetAnnotation(term1,t2,t4);]]: ATisEqual(term1,term2))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm term1=ATremoveAnnotation(t1,t2);]]: ATgetAnnotation(term1, t2) == NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm term1=ATremoveAllAnnotations(t1);]]: ATgetAnnotation(term1,t2) == NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATerm t4=ATsetAnnotation(t1,t2,t3);]] : !ATisEqual(t1,t4) ∧ ATisEqualModuloAnnotations(t1,t4))

(∀: 0≤i<18 ∧ [[ATerm test;test = AT_allocate(3);]]: test)

(∀: s ∈ QSymbol ∧ 0≤i<256 : ATisEqual(ATmakeSymbol(s,i,ATfalse),ATmakeSymbol(s,i,ATfalse)))

(∀: s ∈ USymbol ∧ 0≤i<256 : ATisEqual(ATmakeSymbol(s,i,ATtrue),ATmakeSymbol(s,i,ATtrue)))

(∀: s ∈ QSymbol ∧ 0≤i<256 : ¬ (ATisEqual(ATmakeSymbol(s,i,ATfalse),ATmakeSymbol(s,i,ATtrue))))

(∀: s ∈ QSymbol ∧ 0≤i<256 ∧ 0≤j<7 : ATisEqual(ATmakeSymbol(s,i,ATfalse),ATmakeSymbol(s,j,ATfalse))
∨

i≠j)

(∀: s ∈ USymbol ∧ 0≤i<256 ∧ 0≤j<7 : ATisEqual(ATmakeSymbol(s,i,ATtrue),ATmakeSymbol(s,j,ATtrue))
∨ i≠j)

(∀: s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,1,ATfalse);]]: ATisEqual(ATmakeAppl1(sym,t),
ATmakeAppl1(sym,t)))

(∀: s ∈ QSymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2;
sym = ATmakeSymbol(s, 1, ATfalse); appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl1(sym,t2);]]:
ATisEqual(appl1,appl2) ∨ ¬ATisEqual(t1,t2))

(∀: s ∈ QSymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2;
sym = ATmakeSymbol(s, 1, ATfalse); appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl1(sym,t2);
appl2=ATsetArgument(appl2,t1,0);]]:
ATisEqual(appl1,appl2))

(∀: s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2;
sym = ATmakeSymbol(s, 1, ATfalse); appl1=ATmakeAppl1(sym,t); appl2=ATmakeAppl1(sym,appl1);]]:
¬ATisEqual(appl1,appl2))

(∀: s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2,appl3;
sym = ATmakeSymbol(s, 1, ATfalse);
appl1=ATmakeAppl1(sym,t); appl2=ATmakeAppl1(sym,appl1);
appl3=ATmakeAppl1(sym,t);]]: ¬ATisEqual(appl3,appl2))

(∀: s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym; ATermAppl appl0,appl1,appl2,appl3;
sym = ATmakeSymbol(s, 1, ATfalse);
appl0=ATmakeAppl0(sym); appl1=ATmakeAppl1(sym,t);
appl2=ATmakeAppl1(sym,appl1); appl3=ATsetArgument(appl1, (ATerm)appl0, 0);]]:
¬ATisEqual(appl3,appl2))

(∀: t ∈ Term ∧ [[ATermList l1,l2,l3; l1=ATmakeList1(t);l2=ATmakeList2(t,t);l3=ATmakeList3(t,t,t);]]:
!ATisEmpty(l1) ∧
¬ATisEmpty(l2) ∧ ¬ATisEmpty(l3))

(∀: l ∈ List: ATisEqual(ATelementAt(l,ATgetLength(l)-1),ATgetLast(l)))

(∀: l1 ∈ List ∧ [[ATermList l2=ATgetPrefix(l1);]]: (∀: 0≤i<ATgetLength(l2): ATisEqual(ATelementAt(l1,i),
ATelementAt(l2,i))) ∧ ATgetLength(l1)==ATgetLength(l2)+1)

(∀: l ∈ List ∧ 0≤i<ATgetLength(l) ∧ 0≤j<i ∧ [[ATermList sl=ATgetSlice(l,j,i);]] : (∀:
0≤k<ATgetLength(sl): ATisEqual(ATelementAt(l,k+j),ATelementAt(sl,k))))

(∀: l ∈ List ∧ [[ATerm t=ATgetFirst(l);]]: ATisEqual(t,ATelementAt(l,0)))

(∀: l1 ∈ List ∧ [[ATermList l2=ATgetNext(l1);]]: (∀: 0≤i<ATgetLength(l2): ATisEqual(ATelementAt(l2,i), ATelementAt(l1,i+1))))

(∀: t ∈ Term ∧ l ∈ List ∧ [[ATermList m; m=ATinsert(l,t);]] : (∀: 0≤i<ATgetLength(l): ATisEqual(ATelementAt(m,i+1), ATelementAt(l,i))) ∧ ATisEqual(t,ATelementAt(m,0)) ∧ ATgetLength(m)==ATgetLength(l)+1)

(∀: t ∈ Term ∧ l ∈ List ∧ [[ATermList m; m=ATappend(l,t);]] : (∀: 0≤i<ATgetLength(l): ATisEqual(ATelementAt(m,i),ATelementAt(l,i))) ∧ ATisEqual(t,ATelementAt(m,ATgetLength(l))))

(∀: l1 ∈ List ∧ l2 ∈ List ∧ [[ATermList l=ATconcat(l1,l2);]]: (∀: 0≤i<ATgetLength(l1): ATisEqual(ATelementAt(l,i),ATelementAt(l1,i))) ∧ (∀: 0≤j<ATgetLength(l1): ATisEqual(ATelementAt(l,i+ATgetLength(l1)), ATelementAt(l2,i))) ∧ ATgetLength(l)==ATgetLength(l1)+ATgetLength(l2))

(∀: k ∈ Int ∧ t ∈ Term ∧ [[ATermList l=ATempty; int j; int i=atoi(k); for(j=0;j<i;j++) l=ATinsert(l,t); ;]]: ATgetLength(l)==i)

(∀: l ∈ List ∧ t ∈ Term ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATreplace(l,t,i);]]: (∀: 0≤j<ATgetLength(l): ATisEqual(ATelementAt(l,j),ATelementAt(l2,j)) ∨ (i==j ∧ ATisEqual(ATelementAt(l2,j),t))))

(∀: l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATremoveElementAt(l,i);]]: (∀: 0≤j<i: ATisEqual(ATelementAt(l,j),ATelementAt(l2,j))) ∧ (∀: i≤k<ATgetLength(l2): ATisEqual(ATelementAt(l2,k),ATelementAt(l,k+1))))

(∀: l ∈ List ∧ t ∈ Term ∧ [[ATermList l2=ATremoveAll(l,t);]]: (∀: 0≤i<ATgetLength(l): ¬ATisEqual(ATelementAt(l2,i),t)))

(∀: l ∈ List ∧ t ∈ Term ∧ [[ATermList l2=ATremoveElement(l,t); int index=ATindexOf(l,t,0);]]: (∀: 0≤i<index: ATisEqual(ATelementAt(l,i),ATelementAt(l2,i))) ∧ (∀: index≤j<ATgetLength(l2): ATisEqual(ATelementAt(l2,j),ATelementAt(l,j+1)) ∨ index<0))

(∀: l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATgetTail(l,i); int lengte=ATgetLength(l);]]: (∀: i≤j<ATgetLength(l): ATisEqual(ATelementAt(l2,j-i),ATelementAt(l,j))))

(∀: l ∈ List ∧ tail ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATreplaceTail(l,tail,i);]]: (∀: 0≤j<i: ATisEqual(ATelementAt(l,j),ATelementAt(l2,j))) ∧ (∀: 0≤k<ATgetLength(tail): ATisEqual(ATelementAt(l2,i+k),ATelementAt(tail,k))) ∧ ATgetLength(l2)==i+ATgetLength(tail))

(∀: l ∈ List ∧ t ∈ Term ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATinsertAt(l,t,i);]]: (∀: 0≤j<i: ATisEqual(ATelementAt(l,j),ATelementAt(l2,j))) ∧ ATisEqual(ATelementAt(l2,i),t) ∧ (∀: i≤k<ATgetLength(l): ATisEqual(ATelementAt(l,i),ATelementAt(l2,i+1))))

(∀: t ∈ Term ∧ l ∈ List ∧ 0≤start<ATgetLength(l) ∧ [[int index=ATlastIndexOf(l,t,start);]]:

(∀: index+1≤j<start: !ATisEqual(ATelementAt(l,j),t)) ∧ (index<0 ∨ ATisEqual(ATelementAt(l,index),t)))

(∀: l ∈ List ∧ [[ATermList l2=ATfilter(l,lower3);]]: (∀: 0≤i<ATgetLength(l2):
lower3(ATelementAt(l2,i))))

(∀: s ∈ USymbol ∧ l ∈ List ∧ [[ATermList l2=ATmakeApplList(ATmakeSymbol(s, ATgetLength(l), AT-
true),l);]]:
(∀: 0≤i<ATgetLength(l): ATisEqual(ATgetArguments(l2),l)))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧ t5 ∈ Term
∧ t6 ∈ Term ∧ [[ATermList l1=ATmakeList(6,t1,t2,t3,t4,t5,t6); ATermList l2=ATmakeList6(t1,t2,t3,t4,t5,t6);]]:
ATisEqual(l1,l2))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧ t5 ∈ Term
∧ t6 ∈ Term ∧ [[ATermList l1=ATmakeList(6,t1,t2,t3,t4,t5,t5); ATermList l2=ATmakeList6(t1,t2,t3,t4,t5,t5);]]:
ATisEqual(l1,l2))

(∀: s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧
t5 ∈ Term ∧ t6 ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,6,ATfalse); ATermAppl appl1,appl2;
appl1=ATmakeAppl6(sym,t1,t2,t3,t4,t5,t6);
appl2=ATmakeAppl(sym,t1,t2,t3,t4,t5,t6);]]: ATisEqual(appl1,appl2))

(∀: t ∈ Term : ATdictGet(ATdictCreate(),t)==NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ 0≤k≤10 ∧ 0≤l<10 ∧
[[ATerm d=ATdictCreate(); ATermAppl appl=ATmakeAppl1(ATmakeSymbol("s",1,ATfalse),t1); int i,j;
for(i=0;i<k*10;i++)
d=ATdictPut(d,appl,t3);; d=ATdictPut(d,t1,t2); for(i=0;i<l*10;i++) d=ATdictPut(d,appl,t3); ;]]:
ATisEqual(ATdictGet(d,t1),t2))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ 0≤k≤100 ∧ [[ATerm d=ATdictCreate();
int i; for(i=0;i<k*10;i++) d=ATdictPut(d,t2,t3); ; d=ATdictRemove(d,t1);]]: ATisEqual(ATdictGet(d,t1),NULL))

(∀: i ∈ Int: ATisEqual(ATmake("<int>", i), ATmakeInt(i)))

(∀: r ∈ Real: ATisEqual(ATmake("<real>", r), ATmakeReal(r)))

(∀: s ∈ USymbol: ATisEqual(ATmake("<appl>", s), ATmakeAppl0(ATmakeSymbol(s, 0, ATfalse))))

(∀: s ∈ QSymbol: ATisEqual(ATmake("<str>", s), ATmakeAppl0(ATmakeSymbol(s, 0, ATtrue))))

(∀: t ∈ Term: ATisEqual(ATmake("<placeholder>",t),ATmakePlaceholder(t)))

(∀: t ∈ Term ∧ [[ATermList l1=ATmakeList(1,t); ATermList l2=ATmakeList1(t);]]:
ATisEqual(ATelementAt(l1,0),ATelementAt(l2,0)) ∧ ATgetLength(l1)==1 ∧ ATgetLength(l2)==1)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATermList l1=ATmakeList(2,t1,t2); ATermList l2=ATmakeList2(t1,t2);]]:
ATisEqual(l1,l2))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATermList l1=ATmakeList(3,t1,t2,t3); ATermList
l2=ATmakeList3(t1,t2,t3);]]: ATisEqual(l1,l2))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧
[[ATermList l1=ATmakeList(4,t1,t2,t3,t4); ATermList l2=ATmakeList4(t1,t2,t3,t4);]]: ATisEqual(l1,l2))

(∀: s ∈ USymbol ∧ t1 ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,1,ATfalse); ATermAppl appl1,appl2;
appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl(sym,t1);]]: ATisEqual(appl1,appl2))

(∀: s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,2,ATfalse);
ATermAppl appl1,appl2; appl1=ATmakeAppl2(sym,t1,t2); appl2=ATmakeAppl(sym,t1,t2);]]: ATisEqual(appl1,appl2))

(∀: s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧
[[Symbol sym=ATmakeSymbol(s,3,ATfalse); ATermAppl appl1,appl2; appl1=ATmakeAppl3(sym,t1,t2,t3);
appl2=ATmakeAppl(sym,t1,t2,t3);]]:
ATisEqual(appl1,appl2))

(∀: s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧
[[Symbol sym=ATmakeSymbol(s,4,ATfalse); ATermAppl appl1,appl2; appl1=ATmakeAppl4(sym,t1,t2,t3,t4);
appl2=ATmakeAppl(sym,t1,t2,t3,t4);]]:
ATisEqual(appl1,appl2))

(∀: s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ Term ∧
t5 ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,5,ATfalse); ATermAppl appl1,appl2; appl1=ATmakeAppl5(sym,t1,t2,t3,t4,t5);
appl2=ATmakeAppl(sym,t1,t2,t3,t4,t5);]]: ATisEqual(appl1,appl2))

(∀: in ∈ Int ∧ [[int i,j; ATmatch(ATmake(in), "<int>", &i);j=atoi(in);]]: i==j)

(∀: re ∈ Real ∧ [[double i,j; ATmatch(ATmake(re), "<real>", &i);j=atof(re);]]: i==j)

(∀: q ∈ USymbol ∧ [[char* str;]]: ¬ATmatch(ATmake(q), "<str>", &str))

(∀: q ∈ QSymbol ∧ [[char* str;]]: ¬ATmatch(ATmake(q), "<id>", &str))

(∀: q ∈ QSymbol ∧ [[char* str;]]: ¬ATmatch(ATmake(q), "<appl(1)>", &str))

(∀: q ∈ USymbol ∧ [[char* str;]]: ¬ATmatch(ATmake(q), "<appl(1)>", &str))

(∀: q ∈ QSymbol ∧ [[char* str; char *str2; str2=strcat(q,"(1)");]]: ¬ATmatch(ATmake(str2), "<appl>",
&str))

(∀: q ∈ USymbol ∧ [[char* str; char *str2; str2=strcat(q,"(1)");]]: !ATmatch(ATmake(str2), "<appl>",
&str))

(∀: t1 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); long index=0; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new);]]: ATisEqual(t1,ATindexedSetGetElem(table,index)))

(∀: t1 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); long index=0; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new); ATindexedSetRemove(table,t1);]]: ATindexedSetGetIndex(table,t1)<0)

(∀: t1 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); long index=0; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new); ATindexedSetReset(table);]]: ATindexedSetGetIndex(table,t1)<0)

(∀: t1 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); long index; ATbool new; ATermIndexed-Set table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new); ATermList l=ATindexedSetElements(table);]]: ATisEqual(ATelementAt(l,index),t1))

(∀: k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); ATindexedSetDestroy(table);]]: 1==1)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,t1,t2);]]: ATisEqual(t2,ATtableGet(table,t1)))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,t1,t2); ATtableRemove(table,t1);]]: ATtableGet(table,t1)==NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,t1,t2);]]: ATindexOf(ATtableKeys(table),t1,0)>=0 ∧ ATindexOf(ATtableValues(table),t2,0)>=0)

(∀: k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,ATparse("1"),ATparse("2")); ATtableReset(table);]]: ATisEqual(ATempty,ATtableKeys(table)))

(∀: k ∈ Int ∧ 0≤perc<11 ∧ [[int size=atoi(k); ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,ATparse("1"),ATparse("2")); ATtableDestroy(table);]]: 1==1)

(∀: t1 ∈ Term ∧ [[FILE *file; file = fopen("file.txt","w"); long l=ATwriteToSharedTextFile(t1,file); fclose(file); file = fopen("file.txt", "rb"); ATerm t2=ATreadFromSharedTextFile(file);]]: ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[FILE *file; file = fopen("file.txt","w"); ATbool b=ATwriteToTextFile(t1,file); fclose(file); file = fopen("file.txt", "rb"); ATerm t2=ATreadFromTextFile(file);]]: ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[FILE *file; file = fopen(”file.txt”,”w”); ATbool b=ATwriteToBinaryFile(t1,file); fclose(file);
file =
fopen(”file.txt”, ”rb”); ATerm t2=ATreadFromBinaryFile(file);]]: ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[ATbool b=ATwriteToNamedTextFile(t1,”file”); ATerm t2=ATreadFromNamedFile(”file”);]]:
ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[ATbool b=ATwriteToNamedBinaryFile(t1,”file”); ATerm t2=ATreadFromNamedFile(”file”);]]:
ATisEqual(t1,t2))

(∀: t ∈ Term : ATisEqual(ATreadFromString(ATwriteToString(t)),t))

(∀: t ∈ Term ∧ [[char *ptr; int len; ptr = ATwriteToSharedString(t, &len);]]: ATisEqual(t,ATreadFromSharedString(ptr,
len)))

(∀: t ∈ Term ∧ [[char *ptr; int len; ptr = ATwriteToBinaryString(t, &len);]]: ATisEqual(t,ATreadFromBinaryString(ptr,
len)))

(∀: bl ∈ Blob ∧ [[char *ptr; static ATermBlob b; ATermBlob c; b = ATmakeBlob(strlen(bl), strdup(bl));
printf(”%s”,bl);]] ∧
[[ptr = ATwriteToString((ATerm)b); c = (ATermBlob)ATparse(ptr);]]: strcmp(ATgetBlobData(b), ATget-
BlobData(c)) == 0)

(∀: bl ∈ Blob ∧ [[static ATermBlob b; ATermBlob c; FILE *file; b = ATmakeBlob(strlen(bl), strdup(bl));
file = fopen(”test.blob”, ”wb+”); ATwriteToTextFile((ATerm)b, file); fflush(file); fseek(file, 0, SEEK_SET);
c =
(ATermBlob)ATreadFromTextFile(file);]]: strcmp(ATgetBlobData(b), ATgetBlobData(c)) == 0)

(∀: bl ∈ Blob ∧ [[static ATermBlob b; b = ATmakeBlob(strlen(bl), strdup(bl));]] : ATgetBlobSize(b)==strlen(bl)
∧ strcmp(ATgetBlobData(b),bl)==0)

Appendix D

Initial data file

```
Term: qq(0,54.3,[1,3.4]){[a,1],[b,2],[c,3]} | [[]]{[4.6,<int>]} | [1,4,5]{[s,3]}
 | <real>{[<just-a-test>,"what a test!"]} | 6.66
Int: 1 | 5 | 6 | 200
Real: 1.2345 | 5.501 | 6666666.666
List: [1] | [[],[]] | [[]] | [a,b,c]
 | [1,suc(1),suc(suc(1)),suc(suc(suc(1))),suc(suc(suc(suc(1)))),
suc(suc(suc(suc(suc(1)))))]|[[[[[[[[[[]]]]]]]]]],a,b,c,d,e,f]

USymbol: aa | bb
QSymbol: "application" | "An \" \n \r \t \\ application"

Blob: "abkjjfgjjerjnghksdfgsdfgshosdghujksjngbiwruoijgpkmthuiodjkgpkeihfgjkjwuok3u9024sc"|"a"|"bbb"
```

Appendix E

Quantifications for LKList

We used the following quantifications for testing the LKList library:

(∀: (Int)i ∈ Int ∧ [[int j; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(j=0;j<i;j++) LKInsertAfterInList(mylist, LKLastInList(mylist), &j); ;]]: (∀: 1≤l<i ∧ [[int k; LKGotoInList(mylist, l); LKAdvanceList(mylist); k=LKCurrent(mylist);]]: l+1==k) ∧ (FA: 0≤q<1 ∧ 0≤r<i ∧ [[LKGotoInList(mylist,i); LKAdvanceList(mylist);int m=LKCurrent(mylist);]]: m==1))

(∀: (Int)i ∈ Int ∧ [[int j; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(j=0;j<i;j++) LKInsertAfterInList(mylist, LKLastInList(mylist), &j);;LKClearList(mylist);]]: LKEmptyList(mylist)==(int)NULL)

(∀: (Int)i ∈ Int ∧ [[listptr mylist; LKCreateList(&mylist, sizeof(i));]]: LKEmptyList(mylist)==(int)NULL)

(∀: (Int)i ∈ Int ∧ 1≤j<i+1 ∧ [[int a; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(a=1;a<i+1;a++) LKInsertAfterInList(mylist, LKLastInList(mylist), &a); ;LKDeleteFromList(mylist,j);]]: (∀: 1≤l<j ∧ [[int k1; LKGetFromList(mylist, l, &k1);]]: l==k1) ∧ (∀: j≤m<i ∧ [[int k2; LKGetFromList(mylist, m, &k2);]]: m+1==k2))

(∀: (Int)i ∈ Int ∧ [[int j; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(j=0;j<i;j++) LKInsertAfterInList(mylist, LKLastInList(mylist), &j); ;]] : (∀: 0≤l<i ∧ [[int k; LKGetFromList(mylist, l+1, &k);]]: l==k) ∧ LKLastInList(mylist)==i)

(∀: (Int)i ∈ Int ∧ [[int j; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(j=1;j<i+1;j++) LKInsertAfterInList(mylist, LKLastInList(mylist), &j); ;]]: (∀: 1≤l<i+1 ∧ [[int k; LKGotoInList(mylist, l); k=LKCurrent(mylist);]]: l==k))

(∀: (Int)i ∈ Int ∧ 1≤j<i+1 ∧ [[int a; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(a=1;a<i+1;a++) LKInsertAfterInList(mylist, LKLastInList(mylist), &a); ; a=i+1; LKInsertAfterInList(mylist,j,&a); int q; LKGetFromList(mylist, j+1, &q);]]: (∀: 1≤l<j+1 ∧ [[int k1; LKGetFromList(mylist, l, &k1);]]: l==k1) ∧ (∀: j+2≤m<i ∧ [[int k2; LKGetFromList(mylist, m, &k2);]]: m-1==k2) ∧ q==i+1 ∧ LKEmptyList(mylist)>0)

(∀: (Int)i ∈ Int ∧ 1≤j<i+1 ∧ [[int a; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(a=1;a<i+1;a++)

LKInsertAfterInList(mylist, LKLastInList(mylist), &a);; a=i+1;LKInsertBeforeInList(mylist,j,&a);int q; LKGet-FromList(mylist, j, &q);]]: (∀: 1≤l<j ∧ [[int k1; LKGetFromList(mylist, l, &k1);]]: l==k1) ∧ (∀: j+1≤m<i ∧ [[int k2; LKGetFromList(mylist, m, &k2);]]: m-1==k2) ∧ q==i+1 ∧ LKEmptyList(mylist)>0)

(∀: (Int)i ∈ Int ∧ 1≤j<i+1 ∧ [[int a; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(a=1;a<i+1;a++) LKInsertAfterInList(mylist, LKLastInList(mylist), &a); ;a=i+1; LKPutInList(mylist,j,&a); int q; LKGet-FromList(mylist, j, &q);]]: q==i+1)

(∀: (Int)i ∈ Int ∧ 1≤j<i+1 ∧ [[int a; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(a=1;a<i+1;a++) LKInsertAfterInList(mylist, LKLastInList(mylist), &a); ; LKGotoInList(mylist, j); int k1,k2; LKGetFrom-List(mylist, j, &k1); LKReverseList(mylist); LKGetFromList(mylist, LKCurrent(mylist), &k2);]]: k1==k2+1 ∨ (j==1 ∧ k1==1))

(∀: (Int)i ∈ Int ∧ 2≤j<i+1 ∧ [[int a; listptr mylist; LKCreateList(&mylist, sizeof(i)); for(a=1;a<i+1;a++) LKInsertAfterInList(mylist, LKLastInList(mylist), &a); ; LKZapList(&mylist);]]: mylist==NULL)

Appendix F

Functions of the LibDS framework

AVL tree functions:

```
avlRealTreeHeight
avlNodeKey
avlNodeKeyAsString
avlNewTree
avlClose
avlCloseWithFunction
avlWalk
avlWalkAscending
avlWalkDescending
avlHeight
avlInsert
avlFindNode
avlFind
avlMinimumNode
avlMinimum
avlMaximumNode
avlMaximum
avlNextNode
avlNextNodeByKey
avlPrevNode
avlPrevNodeByKey
avlGetData
avlNodeData
avlNodeUpdateData
avlUpdateData
avlRemoveByKey
avlRemoveNode
avlSetCurrent
avlClearCurrent
```

avlCurrent
avlPrev
avlNext
avlCut
avlFreeNode
avlTotalNodes
avlRootNode
avlLeftNode
avlRightNode
avlNodeHeight
avlCheck

Hashtable functions:

htMake
htMakeHashTable
htClose
htCloseWithFunction
htAdd
htFind
htRemove
htSize
htItems
htEmpty
htConflicts
htWalk

Heap functions:

heapMake
heapNew
heapCloseWithFunction
heapClose
heapMakeIntKeys
heapMakeDoubleKeys
heapMakeFloatKeys
heapMakeStringKeys
heapSetChgFunc
heapInsert
heapDelete
heapFirst
heapElementAt
heapSize
heapEmpty
heapWalk
heapPrintArray
heapPrintTree
heapCheck

Array functions:

```
paMake
paCloseWithFunction
paClose
paAdd
paRemove
paSize
paReplace
paContains
paElementAt
paCurrent
paSetCurrent
paClearCurrent
paFirst
paLast
paNext
paPrev
```

Queue functions:

```
qMake
qClose
qCloseWithFunction
qEnque
qDeque
qWalk
qWalkAscending
qWalkDescending
qLength
qEmpty
qCurrent
qClearCurrent
qSetCurrent
qFirst
qLast
qNext
qPrev
qElemData
qElemInsert
qElemAttach
qElemDetach
qElemFree
qElemFirst
qElemLast
qElemNext
qElemPrev
```

```
qElemCurr
qElemRemove
```

Set functions:

```
setMake
setNew
setClose
setCloseWithFunction
setAdd
setRemove
setFirst
setNext
setSize
setEmpty
setContains
setFind
setUnion
setUnion1
setIntersect
setDifference
setXIntersect
```

Stack functions:

```
stkMake
stkCloseWithFunction
stkClose
stkPush
stkPop
stkPeek
stkSize
stkEmpty
```

Appendix G

Quantifications for the LibDS library

```
(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};
QELEM e;
int wrong=0;
int teller=0;
for (e = qElemFirst(q); e; e = qElemNext(e)) {
if ( *(double*)(qElemData(e))!=number[teller]) wrong++; teller++;
};qClose(q);]]
: wrong==0 ∧ qEmpty(q)==total==0)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};
QELEM e;
int wrong=0;
int teller=total-1;
for (e = qElemLast(q); e; e = qElemPrev(e)) {
if ( *(double*)(qElemData(e))!=number[teller]) wrong++; teller–;
};qClose(q);]]
: wrong==0 )

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
```

```
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};
qSetCurrent(q,qElemFirst(q));]]
: *(double*)qCurrent(q)==number[0]
∧ (∀: 0≤k<1 ∧ [[qSetCurrent(q,qElemLast(q));]]: *(double*)qCurrent(q)==number[total-1] )
∧ (∀: 0≤l<1 ∧ [[qClearCurrent(q);]]: qCurrent(q)==NULL ∧ qNext(q)==qFirst(q) )
∧ (∀: 0≤m<1 ∧ [[qClearCurrent(q);]]: qCurrent(q)==NULL ∧ qPrev(q)==qLast(q) )
)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};
qElemRemove(q,qElemFirst(q));
QELEM e;
int wrong=0;
int teller=1;
for (e = qElemFirst(q); e; e = qElemNext(e)) {
if ( *(double*)(qElemData(e))!=number[teller]) wrong++; teller++;
};qClose(q);]]
: wrong==0)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};
QELEM e,f;
int wrong=0;
e = qElemFirst(q);
while (e) {
f=e;
e = qElemNext(e);
qElemFree(f);
};
]]: 0==0 )

(∀: (Int)total ∈ Int ∧ [[HASHTABLE ht=htMake(-1); char strings[total][total]; int count[total]; int difel=0;
int i; int j;
for(i=0;i<total;i++) count[i]=0;
for(i=0;i<total;i++) { char *temp=strings[i]; int r=random_int(total); for(j=0;j<r;j++) { temp[j]='a'; };
temp[r]='\0'; if (count[r]==0) {count[r]++; difel++;};};
```

for(i=0;i<total;i++) { htAdd(ht,(void*)&strings[i],(void*)&strings[i]); };]]
: htItems(ht)==difel ∧ htEmpty(ht)==(total==0))

(∀: (Int)total ∈ Int ∧ [[HASHTABLE ht=htMakeHashTable(total,hash1,hashcmp1);
HASHTABLE ht2=htMakeHashTable(total,hash3,hashcmp1);int count[total]; int difel=0; int i; int j;
for(i=0;i<total;i++) count[i]=0;
char strings[total][total];
for(i=0;i<total;i++) { char *temp=strings[i]; int r=random_int(total); for(j=0;j<r;j++) { temp[j]='a'; };
temp[r]='\0'; if (count[r]==0) {count[r]++; difel++;};};
for(i=0;i<total;i++) { htAdd(ht,(void*)&strings[i],(void*)&strings[i]); htAdd(ht2,(void*)&strings[i],(void*)&strings[i]);
};]]
: htItems(ht)==difel ∧ htEmpty(ht)==(total==0) ∧ htEmpty(ht2)==(total==0) ∧ htItems(ht2)==difel
∧ htSize(ht)==total ∧
htSize(ht)==total)

(∀: (Int)total ∈ Int ∧ [[HASHTABLE ht=htMakeHashTable(total,hfunc,compare); int number[total]; int
i;
for(i=0;i<total;i++) { number[i]=random_int(total); htAdd(ht,(void*)&number[i],(void*)&number[i]); };]]
: htItems(ht)==total ∧ htEmpty(ht)==(total==0) ∧ (∀: 0≤j<total: htFind(ht,&number[j])!=NULL))

(∀: (Int)total ∈ Int ∧ [[int result1=0; int result2=0; HASHTABLE ht=htMakeHashTable(total,hfunc,compare);
int number[total]; int i;
for(i=0;i<total;i++) { number[i]=random_int(total); htAdd(ht,(void*)&number[i],(void*)&number[i]); re-
sult1+=number[i]; };
static void add_values1(void *pData) { result2+=*(int*)pData; };
htCloseWithFunction(ht,add_values1);]]: result1=result2)

(∀: (Int)total ∈ Int ∧ 0≤j<total ∧ [[HASHTABLE ht=htMakeHashTable(total,hfunc,compare); int num-
ber[total]; int i;
for(i=0;i<total;i++) { number[i]=i; htAdd(ht,(void*)&number[i],(void*)&number[i]); }; htRemove(ht,&number[j]);
]]:
htFind(ht,&number[j])==NULL)

(∀: (Int)total ∈ Int ∧ [[HASHTABLE ht=htMakeHashTable(total,hfunc,compare); int number[total]; int
count[total]; int i; int cf=0;
for(i=0;i<total;i++) count[i]=0;
for(i=0;i<total;i++) { number[i]=random_int(total); htAdd(ht,(void*)&number[i],(void*)&number[i]); count[number[i]]++;
};
for(i=0;i<total;i++) { if(count[i]>1) cf+=count[i]-1; };]]: htConflicts(ht)==cf)

(∀: (Int)total ∈ Int ∧ [[int result1=0; int result2=0; HASHTABLE ht=htMakeHashTable(total,hfunc,compare);
int number[total]; int i;
for(i=0;i<total;i++) { number[i]=random_int(total); htAdd(ht,(void*)&number[i],(void*)&number[i]); re-
sult1+=number[i]; };
static void add_values1(void *pData) { result2+=*(int*)pData; };
htWalk(ht,1,add_values1);]]: result1==result2)

```
(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};]]
: qLength(q)==total ∧
*(double*)qFirst(q)==number[0] ∧ *(double*)qLast(q)==number[total-1] ∧
(∀: 0≤j<total : *(double*)qRemove(q)==number[j] )
)


(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qEnque(q,(void*)&number[i]);
};]]
: qLength(q)==total ∧
*(double*)qFirst(q)==number[0] ∧ *(double*)qLast(q)==number[total-1] ∧
(∀: 0≤j<total : *(double*)qDeque(q)==number[j] )
)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[double result1=0.0; double result2=0.0; double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
result1+=number[i];
};
static void add_values1(void *pData) { result2+=*(double*)pData; };
qCloseWithFunction(q,add_values1);
]]
: result1==result2
)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[int wrong=0; double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};

    int teller=0;
void *e;
```

for (e = qFirst(q); e; e = qNext(q)) { if (*(double*)e!=number[teller]) wrong++; teller++;};]]
: wrong==0)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[int wrong=0; double number[total]; int i; QUEUE q = qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total);
qAdd(q,(void*)&number[i]);
};

int teller=total-1;
void *e;
for (e = qLast(q); e; e = qPrev(q)) { if (*(double*)e!=number[teller]) wrong++; teller–;};]]
: wrong==0)

(∀: (Int)total ∈ Int ∧ [[int result[2]; int i; for(i=0;i<2;i++) result[i]=0;double number[total]; QUEUE q
= qMake();
for(i=0;i<total;i++) {
number[i]=random_num(total); qAdd(q,(void*)&number[i]); result[0]+=number[i];};
static void add_values1(void *pData) { result[1]+=*(double*)pData; };qWalk(q,add_values1);
]]: result[0]==result[1])

(∀: (Int)total ∈ Int ∧ [[int result[2]; int i; for(i=0;i<2;i++) result[i]=0;double number[total+1]; QUEUE
q = qMake();
int index1=0; int index2=total-1; for(i=0;i<total;i++) { number[i]=random_num(total); qAdd(q,(void*)&number[i]);
};
static void comp_doublea(void *pData) { if (*(double*)pData != number[index1]) result[1]++; index1++;
};static void comp_doubled(void
pData) { if ((double*)pData != number[index2]) result[0]++; index2–; };
qWalkAscending(q,comp_doublea);qWalkDescending(q,comp_doubled);]]: result[0]==0 ∧ result[1]==0)

(∀: (Int)total ∈ Int ∧ [[double numbers[total]; int i; STACK stack=stkMake();
for(i=0;i<total;i++) {
numbers[i]=random_num(total);
stkPush(stack,&numbers[i]);
};]]: stkSize(stack)==total ∧ stkEmpty(stack)==0
∧ (∀: 0≤j<total: *(double*)stkPeek(stack)==numbers[total-1-j] ∧ *(double*)stkPop(stack)==numbers[total-
1-j]∧
stkEmpty(stack)==(j==total-1) ∧ stkSize(stack)==total-1-j)
∧ stkPop(stack)==NULL ∧ stkPeek(stack)==NULL ∧ stkSize(stack)==0 ∧ stkEmpty(stack)==1
)

(∀: (Int)total ∈ Int ∧ [[double result1=0.000; double result2=0.000; double numbers[total]; int i; STACK
stack=stkMake();
for(i=0;i<total;i++) {
numbers[i]=random_num(total);

```
stkPush(stack,&numbers[i]);
result1+=numbers[i];
};
static void add_valuess(void *pData) { result2+=*(double*)pData; };
stkCloseWithFunction(stack,add_valuess);
]]: result1==result2
)

(∀: (Int)total ∈ Int ∧ [[double numbers[total]; int i; STACK stack=stkMake();
for(i=0;i<total;i++) {
numbers[i]=random_num(total);
stkPush(stack,&numbers[i]);
};
stkClose(stack);]]: stack==stack)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};]]
: (∀: 0≤j<total ∧ [[double res=*(double*)avlFind(avl,&number[j]);]]: res==number[j])
∧ avlCheck(avl)¿=0 ∧ avlHeight(avl) ¿1.44*log(total)
)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<total ∧
[[int result=0; double number[total+1]; int i; AVLTREE avl = avlNewTree(compare,1,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};
/*number[total]=number[random_int(total-1)];
int temp=number[total]; */]]
: 0==0 ∧ avlCheck(avl)¿=0)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total+1]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};
number[total]=number[random_int(total-1)];
avlSetCurrent(avl,&number[total]);
avlClearCurrent(avl);]]
: avlCurrent(avl)==NULL ∧ avlCheck(avl)¿=0)
```

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total+1]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};
double first=*(double*)avlFirst(avl);]]
: (∀: 0≤k<total: *(double*)avlFind(avl,&number[k])¿=first)
∧ first==*(double*)avlMinimum(avl)
∧ *(double*)avlCurrent(avl)==first
∧ avlCheck(avl)¿=0
)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<1 ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total+1]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};
double last=*(double*)avlLast(avl);]]
: (∀: 0≤k<total: *(double*)avlFind(avl,&number[j])≤last)
∧ last==*(double*)avlMaximum(avl) ∧ avlCheck(avl)¿=0
)

(∀: (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[double result[4]; double number[total+1]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<4;i++) result[i]=0.0;
for(i=0;i<total;i++) {
number[i]=random_num(total); avlAdd(avl,(void*)&number[i],(void*)&number[i]); result[0]+=number[i]; };

    void *e;
for (e = avlFirst(avl); e; e = avlNext(avl)) { result[1]+=*(double*)e; };
for (e = avlLast(avl); e; e = avlPrev(avl)) { result[2]+=*(double*)e; };
static void add_values1(void *pData) { result[3]+=*(double*)pData; }; avlWalk(avl,add_values1,1);]]
: result[0]==result[1] ∧ result[0]=result[2] ∧ result[0]=result[3]
)

(∀: (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result[2]; int i; for(i=0;i<2;i++) result[i]=0;double number[total+1]; AVLTREE avl = avlNewTree(compare,s,10);
double prev=0; double prev1=total*1.1;
for(i=0;i<total;i++) {
number[i]=random_num(total); avlAdd(avl,(void*)&number[i],(void*)&number[i]); };
static void comp_doublea(void *pData) { if (*(double*)pData < prev) result[1]++; ; prev=*(double*)pData;
};
static void comp_doubled(void *pData) { if (*(double*)pData ¿ prev1) result[0]++; ; prev1=*(double*)pData;
};

avlWalkAscending(avl,comp_doublea);avlWalkDescending(avl,comp_doubled);
]]: result[0]==0 ∧ result[1]==0)

(∀: (Int)total ∈ Int ∧ 0≤j<total ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};
j=random_int(total);
double d=0.01;
avlUpdateData(avl,&number[j],&d);]]
: *(double*)avlFind(avl,&number[j])==d
)

(∀: (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int i; AVLTREE avl = avlNewTree(compare,s,10); double number[total]; double teller1=0.0; double teller=0.0;double
teller2=0.0;
for(i=0;i<total;i++) {
number[i]=random_num(total); avlAdd(avl,(void*)&number[i],(void*)&number[i]); teller1+=number[i];};

    AvlNode* node=avlMinimumNode(avl);
while (node!=NULL) {
double temp=*(double*) (NODE(node)-¿an_data);
double temp2=*(double*)avlNodeKey(node);
double temp3=*(double*)avlNodeData(node);
CU_ASSERT(temp==temp2);
CU_ASSERT(temp==temp3);
teller+=temp;
node=avlNextNodeByKey(avl,(void*)&temp);
};

    node=avlMaximumNode(avl);
while (node!=NULL) {
double temp=*(double*) (NODE(node)-¿an_data);
teller2+=temp;
node=avlPrevNodeByKey(avl,(void*)&temp);
};]]: teller==teller1 ∧ teller==teller2)

(∀: (Int)total ∈ Int ∧ [[int result=0; int number[total]; int i; SET set = setMake();
for(i=0;i<total;i++) {
number[i]=random_int(total);
setAdd(set,&number[i]);
};]]
: (∀: 0≤j<setSize(set): setContains(set,&number[j])¿0 ∧ *(int*)setFind(set,&number[j])==number[j])
∧ setSize(set)==total ∧ setEmpty(set)== total==0)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)method ∈ SortMethod ∧
[[int result=0; int number[total]; int i; SET set = setNew(compare_ints,method,1);
for(i=0;i<total;i++) {
number[i]=random_int(total);
setAdd(set,&number[i]);
};]]
: (∀: 0≤j<setSize(set): setContains(set,&number[j])¿0 ∧ *(int*)setFind(set,&number[j])==number[j])
∧ method==1 ∨ setSize(set)==total ∧ method==1 ∨ setEmpty(set)== total==0)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)method ∈ SortMethod ∧
[[int result=0; int number[total]; int i; SET set = setNew(compare_ints,method,1);
for(i=0;i<total;i++) {
number[i]=random_int(total);
setAdd(set,&number[i]);
};
printSet(set);]]
: 0==0)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)method ∈ SortMethod ∧
[[int result=0; int result2=0; int number[total]; int i; SET set = setNew(compare_ints,method,1);
for(i=0;i<total;i++) {
number[i]=random_int(total);
setAdd(set,&number[i]);
};

    void *e;
for (e = setFirst(set); e; e = setNext(set)) {
int t= *(int*) e;
result2+=t;
}

    static void
add_values1(void *pData) {
result+=*(int*)pData;
};
setCloseWithFunction(set,add_values1);]]
: result2==result)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<total ∧ [[int result=0; int number[total]; int i; SET set =
setNew(compare_ints,1,1);
for(i=0;i<total;i++) {
number[i]=random_int(total);
setAdd(set,&number[i]);
};
setRemove(set,&number[j]);

]]
: setContains(set,&number[j])==0 ∧ setFind(set,&number[j])==NULL)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<total ∧ [[int precount=0; int postcount=0; int result=0;
int number[total]; int i; SET set = setNew(compare_ints,0,1);
for(i=0;i<total;i++) {
number[i]=random_int(total);
setAdd(set,&number[i]);
};

for(i=0;i<total;i++) if(number[j]==number[i]) precount++;

setRemove(set,&number[j]);

void *e;
for (e = setFirst(set); e; e = setNext(set)) {
int t= *(int*) e;
if(t==number[j]) postcount++;
};setClose(set);]]: postcount+1==precount)

(∀: (Int)total1 ∈ Int ∧ (Int)total2 ∈ Int ∧ [[int number1[total1];int number2[total2]; int i;
SET set1 = setNew(compare_ints,0,1);
SET set2 = setNew(compare_ints,0,1);for(i=0;i<total1;i++) { number1[i]=random_int(total1); setAdd(set1,&number1[i]);
};
for(i=0;i<total2;i++) { number2[i]=random_int(total2); setAdd(set2,&number2[i]); };
int s1=setSize(set1); int s2=setSize(set2);
SET setunion=setUnion(set1,set2);]]
: (∀: 0≤j<total1 : setContains(setunion,&number1[j]) ∧ setContains(set1,&number1[j]))
∧ (∀: 0≤k<total2 : setContains(setunion,&number2[k]) ∧ setContains(set1,&number2[k]))
∧ setSize(setunion)==total1+total2 ∧ setSize(setunion)==setSize(set1))

(∀: (Int)total1 ∈ Int ∧ (Int)total2 ∈ Int ∧ [[int number1[total1];int number2[total2]; int i;
SET set1 = setNew(compare_ints,0,1);
SET set2 = setNew(compare_ints,0,1);for(i=0;i<total1;i++) { number1[i]=random_int(total1); setAdd(set1,&number1[i]);
};
for(i=0;i<total2;i++) { number2[i]=random_int(total2); setAdd(set2,&number2[i]); };
int s1=setSize(set1); int s2=setSize(set2);
SET setunion=setAppend(set1,set2);]]
: (∀: 0≤j<total1 : setContains(setunion,&number1[j]) ∧ setContains(set1,&number1[j]))
∧ (∀: 0≤k<total2 : setContains(setunion,&number2[k]) ∧ setContains(set1,&number2[k]))
∧ setSize(setunion)==total1+total2 ∧ setSize(setunion)==setSize(set1))

(∀: (Int)total1 ∈ Int ∧ (Int)total2 ∈ Int ∧ (Int)method ∈ SortMethod ∧ [[int number1[total1];int number2[total2]; int i; SET set1 = setNew(compare_ints,method,1);
SET set2 = setNew(compare_ints,method,1);for(i=0;i<total1;i++) {
number1[i]=random_int(total1); setAdd(set1,&number1[i]);

setAdd(set1,&number1[i]); }; for(i=0;i<total2;i++) {
number2[i]=random_int(total2); setAdd(set2,&number2[i]); };
SET setunion=setUnion(set1,set2);
SET setunion2=setAppend(set1,set2);]]
: (∀: 0≤j<total1 : setContains(setunion,&number1[j]) ∧ setContains(setunion2,&number1[j]) ∧
setContains(set1,&number1[j]))
∧ (∀: 0≤k<total2 : setContains(setunion,&number2[k]) ∧ setContains(setunion2,&number2[k]) ∧
setContains(set1,&number2[k]))
∧ setSize(setunion)==setSize(set1) ∧ setSize(setunion2)==setSize(set1))

(∀: (Int)total1 ∈ Int ∧ (Int)total2 ∈ Int ∧ (Int)method ∈ SortMethod ∧ [[int number1[total1];int number2[total2];
int i; SET set1 = setNew(compare_ints,method,1);
SET set2 = setNew(compare_ints,method,1);
for(i=0;i<total1;i++) { number1[i]=random_int(total1); setAdd(set1,&number1[i]); };
for(i=0;i<total2;i++) { number2[i]=random_int(total2); setAdd(set2,&number2[i]); };
SET setunion=setUnion1(set1,set2);
]]: (∀: 0≤j<total1 : setContains(setunion,&number1[j]) ∧ setContains(set1,&number1[j]))
∧ (∀: 0≤k<total2 : setContains(setunion,&number2[k]))
∧ setSize(setunion)==total1+total2 ∨ method==1)

(∀: (Int)total1 ∈ Int ∧ (Int)total2 ∈ Int ∧ (Int)method ∈ SortMethod ∧ [[int number1[total1];int number2[total2];
int i; SET set1 = setNew(compare_ints,method,1);
SET set2 = setNew(compare_ints,method,1);for(i=0;i<total1;i++) {
number1[i]=random_int(total1); setAdd(set1,&number1[i]);
};
for(i=0;i<total2;i++) { number2[i]=random_int(total2); setAdd(set2,&number2[i]); };
SET intersect=setIntersect(set1,set2); SET xintersect=setXIntersect(set1,set2);]]: (∀: 0≤j<total1:
setContains(set2,&number1[j]) == (setContains(intersect,&number1[j])∧!setContains(xintersect,&number1[j]))
)
∧ (∀: 0≤k<total2: setContains(set1,&number2[k]) ==
(setContains(intersect,&number2[k])∧!setContains(xintersect,&number2[k]))))

(∀: (Int)total1 ∈ Int ∧ (Int)total2 ∈ Int ∧ (Int)method ∈ SortMethod ∧ [[int number1[total1];
int number2[total2];
int i; SET set1 = setNew(compare_ints,method,1);
SET set2 = setNew(compare_ints,method,1);for(i=0;i<total1;i++) {
number1[i]=random_int(total1); setAdd(set1,&number1[i]);
};
for(i=0;i<total2;i++) { number2[i]=random_int(total2); setAdd(set2,&number2[i]); };
SET dif=setDifference(set1,set2);]]: (∀: 0≤j<total1 : 1 ∧ (setContains(set1,&number1[j]) ∧
!setContains(set2,&number1[j])) == setContains(dif,&number1[j]))

(∀: (Int)total ∈ Int ∧ [[HEAP heap=heapMake(); char strings[total][total]; int count[total]; int difel=0;
int i; int j; int result=0;

for(i=0;i<total;i++) { char *temp=strings[i]; int r=random_int(total); for(j=0;j<r;j++) { temp[j]='a'; };
temp[r]='\0';};for(i=0;i<total;i++) { heapInsert(heap,(void*)&strings[i],
(void*)&strings[i]); result +=heapCheck(heap);};]]:
(∀: 0≤k<heapSize(heap) ∧ [[int res=0; char *str= malloc(sizeof(char)*total); str=heapElementAt(heap,k);
for(i=0;i<heapSize(heap);i++) { if(strcmp(&strings[i],str)==0) { res++; } };]] : res¿0) ∧ heapSize(heap)==total
∧
(∀: 0≤q<1 ∧ [[heapClose(heap);]]: result==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap = heapNew(compare,i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap,&number[i],&number[i]);
result +=heapCheck(heap);};]]
: (∀: 0≤j<heapSize(heap) ∧ [[double res=0.0; double numper; numper=*(double*)heapElementAt(heap,j);
for(i=0;i<heapSize(heap);i++) { if(number[i]==numper) { res++; } };]]
: res¿0)
∧ heapSize(heap)==total ∧ (∀: 0≤q<1 ∧ [[heapClose(heap);]]: result==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap = heapMakeDoubleKeys(i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap,&number[i],&number[i]);
result +=heapCheck(heap);};]]
: (∀: 0≤j<heapSize(heap) ∧ [[double res=0.0; double numper; numper=*(double*)heapElementAt(heap,j);
for(i=0;i<heapSize(heap);i++)
{ if(number[i]==numper) { res++; } };]]
: res¿0)
∧ heapSize(heap)==total ∧ (∀: 0≤q<1 ∧ [[heapClose(heap);]]: result==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; float number[total]; int i; HEAP heap = heapMakeFloatKeys(i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap,&number[i],&number[i]);
result +=heapCheck(heap);};]]
: (∀: 0≤j<heapSize(heap) ∧ [[double res=0.0; float numper; numper=*(float*)heapElementAt(heap,j);
for(i=0;i<heapSize(heap);i++)
{ if(number[i]==numper) { res++; } };]]
: res¿0)
∧ heapSize(heap)==total ∧ (∀: 0≤q<1 ∧ [[heapClose(heap);]]: result==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; int number[total]; int i; HEAP heap = heapMakeIntKeys(i1,i2,s);
for(i=0;i<total;i++) {

number[i]=random_int(total);
heapInsert(heap,&number[i],&number[i]);
result +=heapCheck(heap);};]]
: (∀: 0≤j<heapSize(heap) ∧ [[double res=0.0; int numper; numper=*(int*)heapElementAt(heap,j);
for(i=0;i<heapSize(heap);i++) { if(number[i]==numper) { res++; } };]]
: res¿0)
∧ heapSize(heap)==total ∧ (∀: 0≤q<1 ∧ [[heapClose(heap);]]: result==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[HEAP heap=heapMakeStringKeys(i1,i2,s); char strings[total][total]; int count[total]; int difel=0; int i; int
j; int result=0;
for(i=0;i<total;i++) { char *temp=strings[i]; int r=random_int(total); for(j=0;j<r;j++) { temp[j]='a'; };
temp[r]='\0';};
for(i=0;i<total;i++) { heapInsert(heap,(void*)&strings[i],(void*)&strings[i]); result +=heapCheck(heap);};]]
: (∀: 0≤k<heapSize(heap) ∧ [[int res=0; char *str= malloc(sizeof(char)*total); str=heapElementAt(heap,k);
for(i=0;i<heapSize(heap);i++) { if(strcmp(&strings[i],str)==0) { res++; } };]]
: res¿0)
∧ heapSize(heap)==total ∧ (∀: 0≤q<1 ∧ [[heapClose(heap);]]: result==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<total ∧ (Int)s ∈ SortMethod ∧
[[int teller=0; int result=0; double number[total]; int i; HEAP heap1 = heapNew(compare,i1,i2,s);
HEAP heap2 = heapNew(compare,i1,i2,0); for(i=0;i<total;i++) { number[i]=random_num(total);
heapInsert(heap1,&number[i],&number[i]); heapInsert(heap2,&number[i],&number[i]); }; heapDelete(heap2,j);]]
: (∀: 0≤k<j ∧ [[int res=heap_double_compare(heapElementAt(heap1,k),heapElementAt(heap2,k)); teller++;]]:
heap_double_compare(heapElementAt(heap1,k),heapElementAt(heap1,k))==0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[double dresult=0.0; double dresult2=0.0; double number[total]; int i; HEAP heap = heapNew(compare,i1,i2,s);
for(i=0;i<total;i++) { number[i]=random_num(total); heapInsert(heap,&number[i],&number[i]); dresult2
+=number[i]; };
static void
add_values4(void *pData) {
dresult+=*(double*)pData;
};
heapCloseWithFunction(heap,add_values4);]]: dresult2==dresult)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap = heapNew(compare,i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap,&number[i],&number[i]);
};]]
: heapElementAt(heap,0)==heapPeekFirst(heap))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap1 = heapNew(compare,i1,i2,s); HEAP heap2 = heap-

New(compare,i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap1,&number[i],&number[i]);
heapInsert(heap2,&number[i],&number[i]);
};
double res=*(double*) heapFirst(heap1);
double res2=*(double*) heapElementAt(heap2,0);
heapDelete(heap2,0);]]
: (∀: 0≤j<heapSize(heap1): *(double*) heapElementAt(heap1,j)==*(double*) heapElementAt(heap2,j))
)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap = heapNew(compare,i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap,&number[i],&number[i]);
};]]: (total==0 == heapEmpty(heap)!=0))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap = heapNew(compare,i1,i2,s);
for(i=0;i<total;i++) {
number[i]=random_num(total);
heapInsert(heap,&number[i],&number[i]);
};heapPrintArray(heap,printfunc);heapPrintTree(heap,printfunc);
]]: 0==0)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int dresult=0.0; int dresult1=0.0; int dresult2=0.0; int number[total]; int i; HEAP heap = heapMakeIntKeys(i1,i2,s);
for(i=0;i<total;i++) { number[i]=random_int(total); heapInsert(heap,&number[i],&number[i]); dresult2 +=number[i]; };
static void
add_valueswalk(int depth, void *pKey, void *pData) {
dresult+=*(int*)pKey;
dresult1+=*(int*)pData;
};
heapWalk(heap,add_valueswalk);]]: dresult2==dresult ∧ dresult==dresult1)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total]; int i; HEAP heap = heapNew(compare,i1,i2,s);
for(i=0;i<total;i++)
number[i]=random_num(total);
static void chgfunc(void* data,int idx) {
double key = *(double*)data; int j; int res=0;
for(j=0;j<total;j++)

```
if(number[j]==key) res++;
if(res==0)
result++;
}

    heapSetChgFunc(heap,chgfunc);

    for (i=0;i<total;i++)
heapInsert(heap,&number[i],&number[i]);]]: result==0)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ [[int boolean; PARRAY pa; pa = paMake(i1,i2);
int i;
for(i=0;i<total;i++) { int afterAdd=paAdd(pa,(void*)i); if(afterAdd!=i) { boolean++; }; };]]
: (∀: 0≤m<total : (int)paElementAt(pa,m)==m)
∧ (∀: 0≤j<1 ∧ [[int laatste=(int)paLast(pa);int current1=(int)paCurrent(pa); int eerste=(int)paFirst(pa);int
current2=(int)paCurrent(pa);]]: paSize(pa)==total ∧ eerste==0 ∧ current1==laatste ∧ current2==eerste
∧ (total==0 ∨
laatste==total-1))
∧ paElementAt(pa,-1)==NULL ∧ paElementAt(pa,total+1)==NULL)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<total ∧ [[PARRAY pa; pa = paMake(i1,i2);
int i; for(i=0;i<total;i++) { paAdd(pa,(void*)i); }; int res=(int)paRemove(pa,j);]]
: res==j ∧ (∀: 0≤k<j: (int)paElementAt(pa,k)==k) ∧ (∀: j≤l<total-1: (int)paElementAt(pa,l)==l+1)
∧ paRemove(pa,-1)==NULL ∧ paRemove(pa,total+1)==NULL)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧
[[int boolean; PARRAY pa; pa = paMake(i1,2); int i; for(i=0;i<total;i++) { paAdd(pa,(void*)i); }; int
current=(int)paCurrent(pa);]]:
current==total-1 ∧ (∀: 1≤l<total ∧ [[int prevnumber=(int)paPrev(pa);]]: prevnumber==total-l-1)
∧ (∀: 1≤m<total ∧ [[int nextnumber=(int)paNext(pa);]]: nextnumber==m)

)(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ [[PARRAY pa; pa = paMake(i1,i2); int i;
for(i=0;i<total;i++) { int afterAdd=paAdd(pa,(void*)i); }; paClearCurrent(pa);]]
: paCurrent(pa)==NULL ∧ paNext(pa)==paFirst(pa)
∧ (∀: 0≤q<1 ∧ [[paClearCurrent(pa);]]: paPrev(pa)==paLast(pa)))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ [[PARRAY pa; pa = paMake(i1,i2); int i;
for(i=0;i<total;i++) { int afterAdd=paAdd(pa,(void*)i); }; paClose(pa);]]
: pa!=NULL)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧
[[PARRAY pa;
pa = paMake(i1,i2);
int i;
int numbers[total+1];
for(i=1;i<total;i++) {
```

```
numbers[i]=i;
int afterAdd=paAdd(pa,&numbers[i]);
};
numbers[total]=total+1;
static int comparee(void *one,void *two) {
int key_one = *(int *) one;
int key_two = *(int *) two;

    int ret = 0;
if (key_one < key_two) {
ret = -1; }
if (key_one ¿ key_two) {
ret = 1; }

    return ret;
}

    int resul = paContains(pa,comparee,&i2);
int resul2 = 0;

    if (i2≤total) {
resul2=1;
};
]]
: resul==resul2 ∧ paContains(pa,comparee,&numbers[total])==0)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧
[[PARRAY pa; pa = paMake(i1,i2); int i; int total1=0; int total2=0;
for(i=0;i<total;i++) { int afterAdd=paAdd(pa,(void*)i); total1+=i; };
static void add_values3(void *pData) { total2+=pData; };
paCloseWithFunction(pa,add_values3);]]: total1==total2)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)total ∈ Int ∧ 0≤j<total ∧
[[PARRAY pa; pa = paMake(i1,i2); int i;
for(i=0;i<total;i++) { int afterAdd=paAdd(pa,(void*)i); };
paReplace(pa,j,0);]]: paElementAt(pa,j)==0 ∧ paReplace(pa,-1,j)==NULL ∧ paReplace(pa,total+1,j)==NULL)

(∀: (Int)i1 ∈ Int ∧ (Int)total ∈ Int ∧ (Int)s ∈ SortMethod ∧
[[int result=0; double number[total+1]; int i; AVLTREE avl = avlNewTree(compare,s,10);
for(i=0;i<total;i++) {
number[i]=random_num(total);
avlAdd(avl,(void*)&number[i],(void*)&number[i]);
};
number[total]=number[random_int(total-1)];
int temp=number[total];
/* avlSetCurrent(avl,&number[total]);
```

```
void *e=avlCut(avl);printf("inputdpsklaar");*/]]
: 0==0 )
```

Appendix H

Occurences of the non terminals in the ATerm grammar

Occs(GATerm,Start)	=	∅
Occs(GATerm,Int)	= {	[aterm−0] ATerm → Int}
Occs(GATerm,Real)	= {	[aterm−1] ATerm → Real}
Occs(GATerm,Appl)	= {	[aterm−2] ATerm → Appl}
Occs(GATerm,List)	= {	[aterm−3] ATerm → List}
Occs(GATerm,Placeholder)	= {	[aterm−4] ATerm → Placeholder}
Occs(GATerm,Blob)	= {	[aterm−5] ATerm → Blob}
Occs(GATerm,Annotations)	= {	[aterm−5] ATerm → Blob, [annotations−6] Annotations → Annotation "," Annotations}
Occs(GATerm,Annotation)	= {	[annotations-0] Annotations → Annotation, [annotations-1] Annotations → Annotation "," Annotations}
Occs(GATerm,FArguments)	= {	[appl−1] Appl → Symbol "(" FArguments ")" [farguments−1] FArguments → ATerm "," FArguments}
Occs(GATerm,LArguments)	= {	[list−1] "[" LArguments "]" [larguments−1] LArguments → ATerm "," LArguments}
Occs(GATerm,ATerm)	= {	[aterm-6] ATerm → ATerm "" Annotations "" [farguments-0] FArguments → ATerm [farguments-1] FArguments → ATerm "," FArguments [larguments-0] LArguments → ATerm [larguments-1] LArguments → ATerm "," LArguments [annotation-0] Annotation → "[" ATerm "," ATerm "]" (first occurrence) [annotation-0] Annotation → "[" ATerm "," ATerm "]" (second occurrence) }

Appendix I

Data file

```
Term: 0
| 1 { [2, -7654321.01234] }
| -4321.00
| -2.22 { ["a"(1),3], [-1.11111,<int>], [[-7654321.01234,[[]]],[2]]}
| (blob)ATerm
| (blob)BLOB { [-0.33,<real>], [4321,a(a())] }
| <placeholder>
| <list> { [[1,"a"(0),(blob)aaaa], (blob)a],[7654321, 0.0],[<blob>,[[]]] }
| "b"(0.33, [[a(3),a(1)],3]) { [(blob)BLOB, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" (1)],
   [aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa([]),"An \ \n \r \t \ application"(1)] }
| "a"( [-4321.00, aaaaa(1)] {[-2.22,4321], [7654321,(blob)BLOB] }, 1)
| "aaaaa"(2.22,"a"(1)) { [[<real>],[2.22]],[ (blob)b,"a"(1)] }
| aaaaa(2,3,<str(<placeholder>)>)
| "An \ \n \r \t \ application"("b"(1),"b"(1),4321.00) { [4321,0.00] }
| "An \ \n \r \t \ application"(7654321)
| aaaaa(7654321.01234) { [<appl(<list>)>,[0]],[ (blob)b,b(1)] }
| "aaaaaa"((blob)bBLOB)
| "An \ \n \r \t \ application" ([<blob>]) { [b(1),b(2)],[b(1),0] }
| "a"(a([1,2]))
| An \ \n \r \t \ application([0,0,0,0]) { [[a(a(1))],[a(1)]] }
| [-7654321.01234,[[]]]
| [(blob)a,<str(<term>)>,a(0)]
| [1,"a"(0),blobaaa] {[2,[1,2]],[(blob)a, <appl(<real>,<placeholder>)>] }
| [[a(3),a(1)],3]
| [-4321.00, aaaaaaa(1)] {[-2.22,4321],[7654321,(blob)aaaaa] }
| [4321,7654321, [<list>,<blob>,<str()>,<real>]] { ["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"(2),
   "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"()], ["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"(3),0] }
| ["An \ \n \r \t \ application"(2), "An \ \n \r \t \ application"(3),-1.11111]{[1,1]}
| [1]
| [-0.33] {[<str(<list>,<blob>,<str()>,<real>)>,2],[ (blob)BLOB, 0.0] }
| [<appl(<blob>,<appl()>)>] {[(blob)b, 0.33] }
```

```
| ["An \ \n \r \t \ application"(4321)]
| [[]] {[1.11111, "An \ \n \r \t \ application"(4321)] }
| [(blob)a]
| [ <int>, <real>, <placeholder>, <list>, <blob>, <term>, <appl(<int>)>, <appl(<real>)>,
 <appl(<placeholder>)>, <appl(<list>)>, <appl(<blob>)>, <appl(<term>)>, <str(<int>)>,
 <str(<real>)>, <str(<placeholder>)>, <str(<list>)>, <str(<blob>)>, <str(<term>)>,
 [<int>], [<real>], [<placeholder>], [<list>], [<blob>], [<term>],
 <appl(<real>,<placeholder>)>, <appl(<list>,<blob>,<str()>,<real>)>,
 <appl(<placeholder>,<appl()>,<list>)>, <appl(<blob>,<appl()>)>,
 <appl(<int>,<int>,<str()>)>, <appl(<term>,<blob>,<blob>)>, <appl(<list>,<list>,<int>)>,
 <appl(<term>,<term>,<term>)>, <str(<real>,<placeholder>)>,
 <str(<list>,<blob>,<str()>,<real>)>, <str(<placeholder>,<appl()>,<list>)>,
 <str(<blob>,<appl()>)>, <str(<int>,<int>,<str()>)>, <str(<term>,<blob>,<blob>)>,
 <str(<list>,<list>,<int>)>, <str(<term>,<term>,<term>)>, [<real>,<placeholder>],
 [<list>,<blob>,<str()>,<real>], [<placeholder>,<appl()>,<list>], [<blob>,<appl()>],
 [<int>,<int>,<str()>], [<term>,<blob>,<blob>], [<term>,<term>,<term>],
 [<list>,<list>,<int>] ]


Int: 0|1|2|3|4321|7654321

Size: 0|1|200|4000

Arity: 1|5|200|255

Real: -7654321.01234|-4321.00|-2.22|-0.33|0|0.33|2.22|4321.00|7654321.01234

List: | [-7654321.01234,[[]]]
| [(blob)a,<str(<term>)>,a(0)]
| [1,"a"(0),(blob)aaa] {[2,[1,2]],[(blob)a, <appl(<real>,<placeholder>)>] }
| [[a(3),a(1)],3]
| [-4321.00, aaaaaaa(1)] {[-2.22,4321],[7654321,(blob)aaaaa] }
| [4321,7654321, [<list>,<blob>,<str()>,<real>]] { ["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"(2),
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"()], ["aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"(3),0] }
| ["An application"(2), "An n"(3),-1.11111]{[1,1]}
| [1]
| [-0.33] {[<str(<list>,<blob>,<str()>,<real>)>,2],[ (blob)BLOB, 0.0] }
| [<appl(<blob>,<appl()>)>] {[(blob)b, 0.33] }
| ["An application"(4321)]
| [[]] {[1.11111, "An \ \n \r \t \ application"(4321)] }
| [(blob)a]
| []
| [[]]

USymbol: a|a|aaaaa|b|An n r t \ application|aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
QSymbol: "a"|"a"|"aaaaa"|"b"|"An n r t \ application"|"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"

Blob: a|a|aaaaa|b|An n r t \ application|aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|
  teksttekstteksttekstteksttekstteksttekstteksttekstteksttekstteksttekstteksttekst
  teksttekstteksttekstteksttekstteksttekstteksttekstteksttekstteksttekstteksttekst

AbortTerm: f("Error",[2,3] [3,4]) | g(5 5) | -0 | <int |
  f(<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,
    <int>,<int>,<int>,<int>,<int>asdfaksdjfhasjkhf) | g(a(1)
```

Appendix J

Complete second set of quantifications

(∀: (Symbol)s ∈ USymbol ∧ [[Symbol sym = ATmakeSymbol(s, 1, ATfalse);AT_print_sym_entries();]]: 1==1)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATbool b=ATisEqual(t1,t2); ATbool c=AT_isDeepEqual(t1,t2);]]: b==c)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATbool b=ATisEqual(t1,t2); ATbool c=AT_isEqual(t1,t2);]]: b==c)

(∀: tm ∈ Term ∧ [[ATerm t[20]; t[0]=tm; t[1]=ATmake("<appl(<term>)>","f",t[0]); t[2]=ATmake("unique-
1");
t[3]=ATmake("unique-2");
t[4]=(ATerm)(t[0]+1);
t[5]=(ATerm)((char *) t[1] + 1);
t[6]=(ATerm)NULL;
t[7]=(ATerm)((MachineWord)testGC);
t[8]=(ATerm)t;
t[9]=(ATerm)"Just a test!";
t[10] = ATsetAnnotation(t[1], t[0], t[3]);
t[11] = ATmake("[<term>,f(<term>)]",tm,tm);
AT_collect(2);]]:
AT_isValidTerm(t[0]) ∧ AT_isValidTerm(t[1]) ∧ AT_isValidTerm(t[2]) ∧ AT_isValidTerm(t[3]) ∧
!AT_isValidTerm(t[4]) ∧ !AT_isValidTerm(t[5]) ∧ !AT_isValidTerm(t[6]) ∧ !AT_isValidTerm(t[7]) ∧
!AT_isValidTerm(t[8]) ∧ !AT_isValidTerm(t[9]) ∧ AT_isValidTerm(t[10]) ∧ AT_isValidTerm(t[11]) ∧
[[AT_markTerm(t[11]);AT_assertMarked(t[11]);AT_assertMarked(t[0]);AT_assertMarked(t[1]);AT_assertUnmarked(t[3]);]]
∧
IS_MARKED(t[0]->header) ∧ IS_MARKED(t[1]->header) ∧ !IS_MARKED(t[2]->header) ∧
!IS_MARKED(t[3]->header) ∧ !IS_MARKED(t[10]->header) ∧ IS_MARKED(t[11]->header) ∧
[[AT_unmarkTerm(t[11]);AT_assertUnmarked(t[11]);AT_assertUnmarked(t[0]);AT_assertUnmarked(t[1]);
AT_assertUnmarked(t[3]);]] ∧ !IS_MARKED(t[11]->header) ∧
[[AT_unmarkAll();int jj;for(jj=0;jj<5;jj++) AT_assertUnmarked(t[jj]);
AT_assertUnmarked(t[10]);AT_assertUnmarked(t[11]);]] ∧
!IS_MARKED(t[0]->header) ∧ !IS_MARKED(t[1]->header) ∧ !IS_MARKED(t[2]->header) ∧
!IS_MARKED(t[3]->header) ∧ !IS_MARKED(t[10]->header) ∧ !IS_MARKED(t[11]->header)

)

(∀: t ∈ Term ∧ [[FILE *file; file = fopen("file.txt","w"); ATfprintf(file,"%t",t); fclose(file);
file = fopen("file.txt", "rb"); ATerm t2=ATreadFromTextFile(file);ATprintf("%t %t\n",t,t2);]]: ATisEqual(t,t2))

(∀: t ∈ Term ∧ [[int c=1456;
char line[1000];char line2[1000];
FILE *file; FILE *file2;v file = fopen("file.txt","w"); ATfprintf(file,"a %c b %d c %i d %o e %u f %x g %X %t",'a',42,43,610,7235,c,c,t);
fclose(file);
file = fopen("file.txt", "rb"); int i=fgets(line, sizeof(line), file);
file2 = fopen("file2.txt","w"); fprintf(file2,"a %c b %d c %i d %o e %u f %x g %X %s",'a',42,43,610,7235,c,c,ATwriteToString(t)); fclose(file2);
file2 = fopen("file2.txt", "rb"); int j=fgets(line2, sizeof(line2), file2);
]]: strcmp(line,line2)==0)

(∀: t ∈ Term ∧ [[int c=1456;
char line[1000];char line2[1000]; char *tekst="tekst";
FILE *file; FILE *file2;
file = fopen("file.txt","w"); ATfprintf(file,"%t h %e i %E j %f k %g l %G m %p n %s",t,663.926, 6663.926, 5.5, 666.666,
666.666, &c,tekst); fclose(file);
file = fopen("file.txt", "rb"); int i=fgets(line, sizeof(line), file);
file2 = fopen("file2.txt","w"); fprintf(file2,"%s h %e i %E j %f k %g l %G m %p n %s",ATwriteToString(t),663.926, 6663.926,
5.5, 666.666, 666.666, &c,tekst);fclose(file2);
file2 = fopen("file2.txt", "rb"); int j=fgets(line2, sizeof(line2), file2);
]]: strcmp(line,line2)==0)

(∀: t ∈ Term ∧ [[int c=1456;
char line[1000];char line2[1000]; char *tekst="tekst";
FILE *file; FILE *file2;
file = fopen("file.txt","w"); ATfprintf(file,"a %c b %d c %i d %o e %u f %x g %X %t h %e i %E j %f k %g l %G m %p n %s",'a',42,43,610,7235,c,c,t,663.926, 6663.926, 5.5, 666.666, 666.666, &c,tekst); fclose(file);
file = fopen("file.txt", "rb"); int i=fgets(line, sizeof(line), file);
file2 = fopen("file2.txt","w"); fprintf(file2,"a %c b %d c %i d %o e %u f %x g %X %s h %e i %E j %f k %g l %G m %p n %s",'a',42,43,610,7235,c,c,
ATwriteToString(t),663.926, 6663.926, 5.5, 666.666, 666.666, &c,tekst);fclose(file2);
file2 = fopen("file2.txt", "rb"); int j=fgets(line2, sizeof(line2), file2);
]]: strcmp(line,line2)==0)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[int c=1456;
char line[1000];char line2[1000]; char *tekst="tekst";
FILE *file; FILE *file2;
file = fopen("file.txt","w"); ATfprintf(file,"a %c b %d c %i d %o e %u f %x g %X %t %t h %e i %E j %f k

%g
l %G m %p n %s”,’a’,42,43,610,7235,c,c,t1,t2,663.926, 6663.926, 5.5, 666.666, 666.666, &c,tekst); fclose(file);
file = fopen(”file.txt”, ”rb”); int i=fgets(line, sizeof(line), file);
file2 = fopen(”file2.txt”,”w”); fprintf(file2,”a %c b %d c %i d %o e %u f %x g %X
%s %s h %e i %E j %f k %g l %G m %p n %s”,
’a’,42,43,610,7235,c,c,ATwriteToString(t1),ATwriteToString(t2),663.926, 6663.926,
5.5, 666.666, 666.666, &c,tekst);fclose(file2);
file2 = fopen(”file2.txt”, ”rb”); int j=fgets(line2, sizeof(line2), file2);
]]: strcmp(line,line2)==0)

(∀: t1 ∈ Term1 ∧ t2 ∈ Term1 ∧ [[int c=1456;
char line[10000];char line2[10000]; char *tekst=”tekst”;
FILE *file; FILE *file2;
file = fopen(”file.txt”,”w”); ATfprintf(file,”%t %t”,t1,t2); fclose(file);
file = fopen(”file.txt”, ”rb”); int itaka=fgets(line, sizeof(line), file);
file2 = fopen(”file2.txt”,”w”); fprintf(file2,”%s %s”,ATwriteToString(t1),
ATwriteToString(t2));fclose(file2);
file2 = fopen(”file2.txt”, ”rb”); int jtaka=fgets(line2, sizeof(line2), file2);
]]: strcmp(line,line2)==0)

(∀: t1 ∈ Term ∧ [[FILE *file; file = fopen(”file.txt”,”w”); long l=ATwriteToSharedTextFile(t1,file); fclose(file);
file =
fopen(”file.txt”, ”rb”); ATerm t2=ATreadFromSharedTextFile(file);]]: ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[FILE *file; file = fopen(”file.txt”,”w”); ATbool b=ATwriteToTextFile(t1,file); fclose(file);
file =
fopen(”file.txt”, ”rb”); ATerm t2=ATreadFromTextFile(file);]]: ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[ATbool b=ATwriteToNamedTextFile(t1,”file”); ATerm t2=ATreadFromNamedFile(”file”);]]:
ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[ATbool b=ATwriteToNamedSharedTextFile(t1,”file”); ATerm t2=ATreadFromNamedFile(”file”);]]:
ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ [[ATbool b=ATwriteToNamedBinaryFile(t1,”file”); ATerm t2=ATreadFromNamedFile(”file”);
ATwriteToNamedBinaryFile(t1,”-”);]]: ATisEqual(t1,t2))

(∀: t ∈ Term : ATisEqual(ATreadFromString(ATwriteToString(t)),t))

(∀: t ∈ Term ∧ [[char *ptr; int len; ptr = ATwriteToSharedString(t, &len);]]: ATisEqual(t,ATreadFromSharedString(ptr,
len)))

(∀: t ∈ Term ∧ [[char *ptr; int len; ptr = ATwriteToBinaryString(t, &len);]]: ATisEqual(t,ATreadFromBinaryString(ptr,
len)))

(∀: t1 ∈ Term ∧ [[FILE *file; file = fopen("file.txt","w"); ATbool b=ATwriteToBinaryFile(t1,file); fclose(file);
file =
fopen("file.txt", "rb"); ATerm t2=ATreadFromBinaryFile(file);]]: ATisEqual(t1,t2))

(∀: t1 ∈ Term ∧ t2 ∈ Term:
ATisEqualModuloAnnotations(t1,t2)==ATisEqual(ATremoveAllAnnotations(t1),ATremoveAllAnnotations(t2)))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATerm aterm=ATremoveAnnotations(t1);
aterm=ATsetAnnotation(aterm,t2,t3)]]: !ATisEqual(ATremoveAnnotations(t1),aterm))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[t1=ATsetAnnotation(t1,t2,t3);t1=ATremoveAnnotation(t1,t2);]]:
ATgetAnnotation(t1,t2)==NULL)

(∀: t1 ∈ Term ∧ t3 ∈ Term: !ATisEqual(t1,ATsetAnnotation(t1,t1,t3)))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATerm c=ATsetAnnotation(ATremoveAnnotations(t1),t2,t3);
c=ATremoveAnnotation(c,t2);]] : ATisEqual(ATremoveAnnotations(t1),c))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[t1=ATsetAnnotation(t1,t2,t3);
t1=ATremoveAnnotation(t1,t2);]] : ATgetAnnotation(t1,t2)==NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATerm c=ATsetAnnotation(ATremoveAnnotations(t1),t2,t3);
c=ATsetAnnotation(ATremoveAnnotations(t1),t2,t3); c=ATremoveAnnotation(c,t2);]] :
ATisEqual(ATremoveAnnotations(t1),c))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ ATerm ∧ [[ATerm term1,term2;
term1=ATsetAnnotation(t1,t2,t3); term1=ATsetAnnotation(term1,t2,t4);]]
: ATisEqual(ATgetAnnotation(term1,t2),t4))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ t4 ∈ ATerm ∧ [[ATerm term1,term2;
term1=ATsetAnnotation(t1,t2,t3); term2=ATsetAnnotation(t1,t2,t4); term1=ATsetAnnotation(term1,t2,t4);]]
: ATisEqual(term1,term2))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm term1=ATremoveAnnotation(t1,t2);]]: ATgetAnnotation(term1,
t2) == NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[t1=ATremoveAllAnnotations(t1);]]: ATgetAnnotation(t1,t2) == NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATerm t4=ATremoveAnnotations(t1);
t4=ATsetAnnotation(t4,t2,t3);]] :
!ATisEqual(ATremoveAnnotations(t1),t4) ∧ ATisEqualModuloAnnotations(ATremoveAnnotations(t1),t4))

(∀: t ∈ Term : ATdictGet(ATdictCreate(), t)==NULL)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ 0≤k<10 ∧

[[ATerm d=ATdictCreate(); ATermAppl appl=ATmakeAppl1(ATmakeSymbol("s",1,ATfalse),t1);
int i,j;
for(i=0;i<k*10;i++)
d=ATdictPut(d,appl,t3);
;
d=ATdictPut(d,t1,t2);
for(i=0;i<k*10;i++)
d=ATdictPut(d,appl,t3);
;]]
: ATisEqual(ATdictGet(d,t1),t2)
)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ 0≤k<10 ∧
[[ATerm d=ATdictCreate();
int i;
for(i=0;i<k*10;i++)
d=ATdictPut(d,t2,t3);
;
d=ATdictRemove(d,t1);]]
: ATisEqual(ATdictGet(d,t1),NULL)
)

(∀: (String)s ∈ QSymbol ∧ (Int)i ∈ Arity : ATisEqual(ATmakeSymbol(s,i,ATfalse),ATmakeSymbol(s,i,ATfalse)))

(∀: (String)s ∈ USymbol ∧ (Int)i ∈ Arity : ATisEqual(ATmakeSymbol(s,i,ATtrue),ATmakeSymbol(s,i,ATtrue)))

(∀: (String)s ∈ QSymbol ∧ (Int)i ∈ Arity : !ATisEqual(ATmakeSymbol(s,i,ATfalse),ATmakeSymbol(s,i,ATtrue)))

(∀: (String)s ∈ QSymbol ∧ (Int)i ∈ Arity ∧ 0≤j<7 : ATisEqual(ATmakeSymbol(s,i,ATfalse),ATmakeSymbol(s,j,ATfalse))
∨ i≠j)

(∀: (String)s ∈ USymbol ∧ (Int)i ∈ Arity ∧ 0≤j<7 : ATisEqual(ATmakeSymbol(s,i,ATtrue),
ATmakeSymbol(s,j,ATtrue)) ∨ i≠j)

(∀: (Symbol)s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,1,ATfalse);]]:
ATisEqual(ATmakeAppl1(sym,t),ATmakeAppl1(sym,t)))

(∀: (Symbol)s ∈ QSymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2;
sym = ATmakeSymbol(s, 1, ATfalse);
appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl1(sym,t2);]]: ATisEqual(appl1,appl2) ∨ !ATisEqual(t1,t2))

(∀: (Symbol)s ∈ QSymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2;
sym = ATmakeSymbol(s, 1, ATfalse);
appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl1(sym,t2); appl2=ATsetArgument(appl2,t1,0);]]
: ATisEqual(appl1,appl2))

(∀: (Symbol)s ∈ QSymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧
[[Symbol sym; ATermAppl appl1,appl2; sym = ATmakeSymbol(s, 1, ATfalse);
appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl1(sym,t2); appl2=ATsetAnnotation(appl2,t1,t2);
appl2=ATsetArgument(appl2,t1,0);]]: !ATisEqual(appl1,appl2))

(∀: (Symbol)s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2; sym = ATmakeSymbol(s, 1, ATfalse);
appl1=ATmakeAppl1(sym,t); appl2=ATmakeAppl1(sym,appl1);]]: !ATisEqual(appl1,appl2))

(∀: (Symbol)s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym; ATermAppl appl1,appl2,appl3;
sym = ATmakeSymbol(s, 1, ATfalse);
appl1=ATmakeAppl1(sym,t); appl2=ATmakeAppl1(sym,appl1); appl3=ATmakeAppl1(sym,t);]]: !ATisEqual(appl3,appl2))

(∀: (Symbol)s ∈ QSymbol ∧ t ∈ Term ∧ [[Symbol sym; ATermAppl appl0,appl1,appl2,appl3;
sym = ATmakeSymbol(s, 1, ATfalse);
appl0=ATmakeAppl0(sym); appl1=ATmakeAppl1(sym,t); appl2=ATmakeAppl1(sym,appl1);
appl3=ATsetArgument(appl1, (ATerm)appl0, 0);]]:
!ATisEqual(appl3,appl2))

(∀: t ∈ Term ∧ [[ATermList l1,l2,l3; l1=ATmakeList1(t);l2=ATmakeList2(t,t);l3=ATmakeList3(t,t,t);]]:
!ATisEmpty(l1) ∧
!ATisEmpty(l2) ∧ !ATisEmpty(l3))

(∀: (ATermList)l ∈ List: ATisEmpty(l)==ATgetLength(l)==0)

(∀: (ATermList)l ∈ List: ATisEqual(ATelementAt(l,ATgetLength(l)-1),ATgetLast(l)))

(∀: (ATermList)l1 ∈ List ∧ [[ATermList l2=ATgetPrefix(l1);]]: (∀: 0≤i<ATgetLength(l2) :
ATisEqual(ATelementAt(l1,i),ATelementAt(l2,i))) ∧ ATisEmpty(l1) ∨ ATgetLength(l1)==ATgetLength(l2)+1)

(∀: (ATermList)l ∈ List ∧ 0≤i<ATgetLength(l) ∧ 0≤j<i ∧ [[ATermList sl=ATgetSlice(l,j,i);]] :
(∀: 0≤k<ATgetLength(sl) :
ATisEqual(ATelementAt(l,k+j),ATelementAt(sl,k))))

(∀: (ATermList)l ∈ List: ATisEqual(ATmakeList0(),ATgetSlice(l,0,0)) ∧
ATisEqual(ATremoveAnnotations(l),ATgetSlice(l,0,ATgetLength(l))))

(∀: (ATermList)l ∈ List ∧ [[ATerm t=ATgetFirst(l);]]: ATisEqual(t,ATelementAt(l,0)))

(∀: (ATermList)l1 ∈ List: (∀: 0≤i<ATgetLength(l1) ∧ [[ATermList l2=ATgetNext(l1);]]:
ATisEqual(ATelementAt(l2,i),ATelementAt(l1,i+1))))

(∀: t ∈ Term ∧ (ATermList)l ∈ List ∧ [[ATermList m; m=ATinsert(l,t);]] : (∀: 0≤i<ATgetLength(l):
ATisEqual(ATelementAt(m,i+1),ATelementAt(l,i))) ∧ ATisEqual(t,ATelementAt(m,0)) ∧
ATgetLength(m)==ATgetLength(l)+1)

(∀: t ∈ Term ∧ (ATermList)l ∈ List ∧ [[ATermList m; m=ATappend(l,t);]] : (∀: 0≤i<ATgetLength(l):
ATisEqual(ATelementAt(m,i),ATelementAt(l,i))) ∧ ATisEqual(t,ATelementAt(m,ATgetLength(l))))

(∀: (ATermList)l1 ∈ List ∧ (ATermList)l2 ∈ List ∧ [[ATermList l=ATconcat(l1,l2);]]:
(∀: 0≤i<ATgetLength(l1) : ATisEqual(ATelementAt(l,i),ATelementAt(l1,i)))
∧ (∀: 0≤j<ATgetLength(l1) : ATisEqual(ATelementAt(l,i+ATgetLength(l1)),ATelementAt(l2,i)))
∧ ATgetLength(l)==ATgetLength(l1)+ATgetLength(l2)
)

(∀: (Int)k ∈ Size ∧ t ∈ Term ∧
[[ATermList l=ATempty; int j; for(j=0;j<k;j++) l=ATinsert(l,t); ;]]
: ATgetLength(l)==k
)

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATreplace(l,t,i);]]
: (∀: 0 ≤j<ATgetLength(l): ATisEqual(ATelementAt(l,j),ATelementAt(l2,j)) ∨ (i==j ∧ ATisEqual(ATelementAt(l2,j),t))))

(∀: (ATermList)l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATremoveElementAt(l,i);]]
: (∀: 0≤j<i: ATisEqual(ATelementAt(l,j),ATelementAt(l2,j))) ∧
(∀: i≤k<ATgetLength(l2): ATisEqual(ATelementAt(l2,k),ATelementAt(l,k+1)))
)

(∀: (ATermList)l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATremoveElementAt(l,i);ATermList
l3=ATremoveElement(l,ATelementAt(l,i));]]
: ATisEqual(l2,l3) ∧ ATgetLength(l2)==ATgetLength(l)-1)

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ [[ATermList l2=ATremoveAll(l,t);]]
: (∀: 0≤i<ATgetLength(l): !ATisEqual(ATelementAt(l2,i),t)))

(∀: (ATermList)l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATremoveAll(l,ATelementAt(l,i));]]
: (∀: 0≤j<ATgetLength(l2): !ATisEqual(ATelementAt(l2,j),ATelementAt(l,i))))

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ [[ATermList l2=ATremoveElement(l,t); int index=ATindexOf(l,t,0);]]
: (∀: 0≤i<index: ATisEqual(ATelementAt(l,i),ATelementAt(l2,i)))
∧ (∀: index≤j<ATgetLength(l2): ATisEqual(ATelementAt(l2,j),ATelementAt(l,j+1)) ∨ index<0))

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATinsertAt(l,t,i);
ATbool b= (ATindexOf(l2,t,0)==i ∧ ATindexOf(l2,t,i+1)==-1 ∧ ATindexOf(l2,t,i+1-ATgetLength(l))==-
1 ∨ ATlastIndexOf(l2,t,0)>i);]]:b)

(∀: (ATermList)l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATgetTail(l,i); int lengte=ATgetLength(l);]]
: (∀: i≤j<ATgetLength(l): ATisEqual(ATelementAt(l2,j-i),ATelementAt(l,j))))

(∀: (ATermList)l ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[int index=-1*i; ATermList l2=ATgetTail(l,index); int

uit=ATgetLength(l)+index; ATermList l3=ATgetTail(l,uit);]]: ATisEqual(l2,l3))

(∀: (ATermList)l ∈ List ∧ (ATermList)tail ∈ List ∧ 0≤i<ATgetLength(l) ∧ [[ATermList
l2=ATreplaceTail(l,tail,i); ATermList l3=ATreplaceTail(l,tail,i-ATgetLength(l));]]
: (∀: 0≤j<i: ATisEqual(ATelementAt(l,j),ATelementAt(l2,j)))
∧ (∀: 0≤k<ATgetLength(tail): ATisEqual(ATelementAt(l2,i+k),ATelementAt(tail,k)))
∧ ATgetLength(l2)==i+ATgetLength(tail)
∧ ATisEqual(l2,l3)
)

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ 0≤i<ATgetLength(l) ∧ [[ATermList l2=ATinsertAt(l,t,i);]]
: (∀: 0≤j<i: ATisEqual(ATelementAt(l,j),ATelementAt(l2,j)))
∧ ATisEqual(ATelementAt(l2,i),t)
∧ (∀: i≤k<ATgetLength(l): ATisEqual(ATelementAt(l,i),ATelementAt(l2,i+1)))
)

(∀: t ∈ Term ∧ (ATermList)l ∈ List ∧ 0≤start<ATgetLength(l) ∧ [[int index=ATlastIndexOf(l,t,start);int
index2=ATlastIndexOf(l,t,start-ATgetLength(l));]]
: (∀: index+1≤j<start: !ATisEqual(ATelementAt(l,j),t))
∧ (index<0 ∨ ATisEqual(ATelementAt(l,index),t))
∧ index==index2
)

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ [[ATermList l2=ATinsert(l,t); l2=ATsort(l2,ATcompare);]]
: (∀: 0≤i<ATgetLength(l): i==0 ∨ ATcompare(ATelementAt(l2,i-1),ATelementAt(l2,i))≤0)
)

(∀: (ATermList)l ∈ List ∧ [[ATermList l2=ATsort(l,ATcompare);]]
: (∀: 0≤i<ATgetLength(l): i==0 ∨ ATcompare(ATelementAt(l2,i-1),ATelementAt(l2,i))≤0)
)

(∀: (ATermList)l ∈ List ∧ [[ATermList l2=ATfilter(l,lower3);]]
: (∀: 0≤i<ATgetLength(l2): lower3(ATelementAt(l2,i)))
)

(∀: (String)s ∈ USymbol ∧ (ATermList)l ∈ List ∧ [[ATermList l2=ATmakeApplList(ATmakeSymbol(s,
ATgetLength(l), ATtrue),l);]]
: (∀: 0≤i<ATgetLength(l): ATisEqual(ATgetArguments(l2),ATremoveAnnotations(l)))
)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATermList l1=ATmakeList(6,t1,t2,t3,t1,t2,t3);
ATermList l2=ATmakeList6(t1,t2,t3,t1,t2,t3);]]: ATisEqual(l1,l2))

(∀: (Int)i ∈ Int ∧ [[ATerm ATint=ATmake("<int>", i);]]: ATisEqual(ATint, ATmakeInt(i)) ∧
ATisEqual(ATint,ATmakeTerm(AT_getPattern("<int>"),i)) ∧ [[int res; ATmatch(ATint,"<int>",&res);]] ∧
res==i)

(∀: (Float)f ∈ Real ∧ [[ATerm ATfloat=ATmake("<real>", f);]]: ATisEqual(ATfloat, ATmakeReal(f)) ∧
ATisEqual(ATfloat,ATmakeTerm(AT_getPattern("<real>"),f)) ∧ [[double res; ATmatch(ATfloat,"<real>",&res);]]
∧ res==f)

(∀: (String)s ∈ USymbol ∧ [[ATerm ATsym=ATmake("<appl>",s);]]: ATisEqual(ATsym, ATmakeAppl0(ATmakeSymbol(s,
0, ATfalse)))
∧ [[char *res; ATmatch(ATsym,"<appl>",&res);]] ∧ strcmp(res,s)==0)

(∀: (String)s ∈ QSymbol ∧ [[ATerm ATsym=ATmake("<str>",s);]]: ATisEqual(ATsym, ATmakeAppl0(ATmakeSymbol(s,
0, ATtrue))) ∧
[[char *res; ATmatch(ATsym,"<str>",&res);]] ∧ strcmp(res,s)==0)

(∀: t ∈ Term ∧ [[ATerm ATpl=ATmake("<placeholder>",t);]] : ATisEqual(ATpl,ATmakePlaceholder(t)) ∧
ATisEqual(ATpl,ATmakeTerm(AT_getPattern("<placeholder>"),t)))

(∀: t ∈ Term ∧ [[ATermList l1=ATmakeList(1,t); ATermList l2=ATmakeList1(t);]]:
ATisEqual(ATelementAt(l1,0),ATelementAt(l2,0))
∧ ATgetLength(l1)==1 ∧
ATgetLength(l2)==1 ∧ [[ATermList tlist=ATmakeList1(t); ATermList res;
ATmatch((ATerm)tlist,"[<list>]",&res);]] ∧ ATisEqual(ATelementAt(res,0),t))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATermList l1=ATmakeList(2,t1,t2); ATermList l2=ATmakeList2(t1,t2);]]:
ATisEqual(l1,l2)
∧ [[ATermList res; ATmatch((ATerm)l1,"[<list>]",&res);]] ∧ ATisEqual(t1,ATelementAt(res,0)) ∧
ATisEqual(t2,ATelementAt(res,1)) ∧ ATgetLength(res)==2)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATermList l1=ATmakeList(3,t1,t2,t3); ATermList
l2=ATmakeList3(t1,t2,t3);]]: ATisEqual(l1,l2) ∧ [[ATermList res; ATmatch((ATerm)l1,"[<list>]",&res);]] ∧
ATisEqual(t1,ATelementAt(res,0)) ∧ ATisEqual(t2,ATelementAt(res,1)) ∧ ATisEqual(t3,ATelementAt(res,2))
∧ ATgetLength(res)==3)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATermList l1=ATmakeList(4,t1,t2,t3,t1); ATermList
l2=ATmakeList4(t1,t2,t3,t1);]]: ATisEqual(l1,l2) ∧ [[ATermList res; ATmatch((ATerm)l1,"[<list>]",&res);]]
∧
ATisEqual(t1,ATelementAt(res,0)) ∧ ATisEqual(t2,ATelementAt(res,1)) ∧ ATisEqual(t3,ATelementAt(res,2))
∧
ATisEqual(t1,ATelementAt(res,3)) ∧ ATgetLength(res)==4)

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,1,ATfalse); ATermAppl appl1,appl2;
appl1=ATmakeAppl1(sym,t1); appl2=ATmakeAppl(sym,t1);]]: ATisEqual(appl1,appl2) ∧ [[char *res; ATerm
aterm[2];
ATmatch(appl1,"<appl(<term>)>",&res,&aterm[0]);]] ∧ strcmp(res,s)==0 ∧ ATisEqual(aterm[0],t1))

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ [[Symbol sym=ATmakeSymbol(s,2,ATfalse); ATermAppl

appl1,appl2; appl1=ATmakeAppl2(sym,t1,t2); appl2=ATmakeAppl(sym,t1,t2);]]: ATisEqual(appl1,appl2)
∧ [[char *res; ATerm aterm[2];
ATmatch(appl1,"<appl(<term>,<term>)>",&res,&aterm[0],&aterm[1]);]] ∧ strcmp(res,s)==0 ∧ ATisEqual(aterm[0],t1)
∧ ATisEqual(aterm[1],t2))

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[Symbol
sym=ATmakeSymbol(s,3,ATfalse); ATermAppl appl1,appl2; appl1=ATmakeAppl3(sym,t1,t2,t3);
appl2=ATmakeAppl(sym,t1,t2,t3);]]: ATisEqual(appl1,appl2) ∧ [[char *res; ATerm aterm[5];
ATmatch(appl1,"<appl(<term>,<term>,<term>)>",&res,&aterm[0],&aterm[1],&aterm[2]);]]
∧ strcmp(res,s)==0 ∧ ATisEqual(aterm[0],t1) ∧ ATisEqual(aterm[1],t2) ∧ ATisEqual(aterm[2],t3))

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[Symbol
sym=ATmakeSymbol(s,4,ATfalse); ATermAppl appl1,appl2; appl1=ATmakeAppl4(sym,t1,t2,t3,t1);
appl2=ATmakeAppl(sym,t1,t2,t3,t1);]]: ATisEqual(appl1,appl2) ∧ [[char *res; ATerm aterm[5];
ATmatch(appl1,"<appl(<term>,<term>,<term>,<term>)>",&res,&aterm[0],&aterm[1],&aterm[2],&aterm[3]);]]
∧
strcmp(res,s)==0 ∧ ATisEqual(aterm[0],t1) ∧ ATisEqual(aterm[1],t2) ∧ ATisEqual(aterm[2],t3) ∧ ATisEqual(aterm[3],t1))

(∀: (String)bl ∈ Blob ∧ [[static ATermBlob b; ATermBlob c; b = ATmakeBlob(strlen(bl), strdup(bl));
c = (ATermBlob)ATmake("<blob>",strlen(bl),strdup(bl));]]: strcmp(ATgetBlobData(b), ATgetBlobData(c))
== 0 ∧
strcmp(ATgetBlobData(b),bl) == 0 ∧ strlen(bl)==ATgetBlobSize(b) ∧ strlen(bl)==ATgetBlobSize(c))

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[Symbol
sym=ATmakeSymbol(s,5,ATfalse); ATermAppl appl1,appl2; appl1=ATmakeAppl5(sym,t1,t2,t3,t1,t2);
appl2=ATmakeAppl(sym,t1,t2,t3,t1,t2);]]:
ATisEqual(appl1,appl2))

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[Symbol
sym=ATmakeSymbol(s,6,ATfalse); ATermAppl appl1,appl2; appl2=ATmakeAppl(sym,t1,t2,t3,t1,t2,t3);
appl1=ATmakeAppl6(sym,t1,t2,t3,t1,t2,t3);]]:
ATisEqual(appl1,appl2))
(∀: (String)s ∈ QSymbol ∧ [[ATerm ATsym=ATmake("<id>",s);char *res; ATmatch(ATsym,"<id>",&res);]]:
strcmp(res,s)==0)

(∀: (String)in ∈ Int ∧ [[int i,j; ATmatch(ATmake(in), "<int>", &i);j=atoi(in);]]: i==j)

(∀: (String)re ∈ Real ∧ [[double i,j; ATmatch(ATmake(re), "<real>", &i);j=atof(re);]]: i==j)

(∀: (String)q ∈ USymbol ∧ [[char* str;]]: !ATmatch(ATmake(q), "<str>", &str))

(∀: (String)q ∈ QSymbol ∧ [[char* str;]]: !ATmatch(ATmake(q), "<id>", &str))

(∀: (String)q ∈ QSymbol ∧ [[char* str;]]: !ATmatch(ATmake(q), "<appl(1)>", &str))

(∀: (String)q ∈ USymbol ∧ [[char* str;]]: !ATmatch(ATmake(q), "<appl(1)>", &str))

(∀: (String)q ∈ QSymbol ∧ [[char* str; char *str2; str2=strcat(q,"(1)");]]: !ATmatch(ATmake(str2), "<appl>", &str))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ (Int)size ∈ Size ∧ 0≤perc<11 ∧ [[ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,t1,t2);]]: ATisEqual(t2,ATtableGet(table,t1)) ∧ [[ATtableDestroy(table)]])

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ (Int)size ∈ Size ∧ 0≤perc<11 ∧ [[ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,t1,t2); ATtableRemove(table,t1);]]: ATtableGet(table,t1)==NULL ∧ [[ATtableDestroy(table)]])

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ (Int)size ∈ Size ∧ 0≤perc<11 ∧ [[ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,t1,t2);]]: ATindexOf(ATtableKeys(table),t1,0)>=0 ∧ ATindexOf(ATtableValues(table),t2,0)>=0 ∧ [[ATtableDestroy(table)]])

(∀: (Int)size ∈ Size ∧ 0≤perc<11 ∧ [[ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,ATparse("1"),ATparse("2")); ATtableReset(table);]]: ATisEqual(ATempty,ATtableKeys(table)) ∧ [[ATtableDestroy(table)]])

(∀: (Int)size ∈ Size ∧ 0≤perc<11 ∧ [[ATermTable table=ATtableCreate(size,(perc*10)); ATtablePut(table,ATparse("1"),ATparse("2")); ATtableDestroy(table);]]: 1==1)

(∀: t1 ∈ Term ∧ (Int)size ∈ Size ∧ 9≤perc<10 ∧
[[long index=0; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new);]]
: ATisEqual(t1,ATindexedSetGetElem(table,index)) ∧ [[ATindexedSetDestroy(table);]])

(∀: t1 ∈ Term ∧ (Int)size ∈ Size ∧ 0≤perc<10 ∧ [[long index=0; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new); ATindexedSetRemove(table,t1);]]
: ATindexedSetGetIndex(table,t1)<0 ∧ [[ATindexedSetDestroy(table);]])

(∀: t1 ∈ Term ∧ (Int)size ∈ Size ∧ 0≤perc<10 ∧ [[long index=0; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new); ATindexedSetReset(table);]]
: ATindexedSetGetIndex(table,t1)<0 ∧ [[ATindexedSetDestroy(table);]])

(∀: t1 ∈ Term ∧ (Int)size ∈ Size ∧ 0≤perc<10 ∧ [[long index; ATbool new; ATermIndexedSet table=ATindexedSetCreate(size,(perc*10)); index=ATindexedSetPut(table,t1, &new); ATermList l=ATindexedSetElements(table);]]
: ATisEqual(ATelementAt(l,index),t1) ∧ [[ATindexedSetDestroy(table);]])
(∀: (Int)size ∈ Size ∧ 0≤perc<10 ∧ [[ATermIndexedSet

table=ATindexedSetCreate(size,(perc*10));ATindexedSetDestroy(table);]]
: 1==1)

(∀: 0≤k<1 ∧ [[ATermIndexedSet set=ATindexedSetCreate(2,75);ATbool new;]] ∧ 0≤i<1000 ∧
[[ATindexedSetPut(set,ATmake("f(<int>)",i),&new);ATindexedSetPut(set,ATmake("f(<int>)",i),&new);]]:
ATindexedSetGetIndex(set,ATmake("f(<int>)",i))>-1
)

(∀: (Int)i ∈ Int ∧
[[ATerm ATint=ATmake("<int>", i);]]:
ATisEqual(ATint, ATmakeInt(i)) ∧ ATisEqual(ATint,ATmakeTerm(AT_getPattern("<int>"),i)) ∧
[[int res; ATmatch(ATint,"<int>",&res);]] ∧
res==i)

(∀: (Float)f ∈ Real ∧ [[ATerm ATfloat=ATmake("<real>", f);]]: ATisEqual(ATfloat, ATmakeReal(f)) ∧
ATisEqual(ATfloat,ATmakeTerm(AT_getPattern("<real>"),f)) ∧ [[double res; ATmatch(ATfloat,"<real>",&res);]]
∧ res==f)

(∀: (String)s ∈ USymbol ∧ [[ATerm ATsym=ATmake("<appl>",s);]]: ATisEqual(ATsym, ATmakeAppl0(
ATmakeSymbol(s, 0, ATfalse))) ∧ [[char *res; ATmatch(ATsym,"<appl>",&res);]] ∧ strcmp(res,s)==0)

(∀: (String)s ∈ QSymbol ∧ [[ATerm ATsym=ATmake("<str>",s);]]: ATisEqual(ATsym, ATmakeAppl0(
ATmakeSymbol(s, 0, ATtrue))) ∧ [[char *res; ATmatch(ATsym,"<appl>",&res);]] ∧ strcmp(res,s)==0)

(∀: t ∈ Term ∧ [[ATerm ATpl=ATmake("<placeholder>",t);]] :
ATisEqual(ATpl,ATmakePlaceholder(t)) ∧ ATisEqual(ATpl,ATmakeTerm(AT_getPattern("<placeholder>"),t)))

(∀: t ∈ Term ∧ [[ATermList l1=ATmakeList(1,t); ATermList l2=ATmakeList1(t);]]:
ATisEqual(ATelementAt(l1,0),ATelementAt(l2,0))
∧ ATgetLength(l1)==1 ∧ ATgetLength(l2)==1 ∧ [[ATermList tlist=ATmakeList1(t); ATermList res;
ATmatch((ATerm)tlist,"[<list>]",&res);]] ∧ ATisEqual(ATelementAt(res,0),t))

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATermList l1=ATmakeList(2,t1,t2);
ATermList l2=ATmakeList2(t1,t2);]]: ATisEqual(l1,l2) ∧
[[ATermList res;
ATmatch((ATerm)l1,"[<list>]",&res);]] ∧ ATisEqual(t1,ATelementAt(res,0)) ∧
ATisEqual(t2,ATelementAt(res,1)) ∧ ATgetLength(res)==2)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧ [[ATermList l1=ATmakeList(3,t1,t2,t3); ATermList
l2=ATmakeList3(t1,t2,t3);]]: ATisEqual(l1,l2) ∧ [[ATermList res; ATmatch((ATerm)l1,"[<list>]",&res);]] ∧
ATisEqual(t1,ATelementAt(res,0)) ∧ ATisEqual(t2,ATelementAt(res,1)) ∧ ATisEqual(t3,ATelementAt(res,2))
∧ ATgetLength(res)==3)

(∀: (String)b ∈ Blob ∧ [[ATermBlob atb = ATmake("<blob>",strlen(b),strdup(b));]] :
ATisEqual(ATmakeBlob(strlen(b),ATgetBlobData(atb)),atb) ∧ [[int size; char *string; string=malloc(sizeof(char)*100);
ATmatch(atb,"<blob>",&size,&string);]] ∧ size==strlen(b) ∧ strcmp(b,string)==0)

(∀: (String)s ∈ QSymbol ∧ (Int)i ∈ Int ∧
[[ATerm aterm=ATmake("<str(<int>)>",s,i);
char *res0; int res1;
ATmatch(aterm,"<str(<int>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ res1==i)

(∀: (String)s ∈ QSymbol ∧ (Float)r ∈ Real ∧
[[ATerm aterm=ATmake("<str(<real>)>",s,r);
char *res0; double res1;
ATmatch(aterm,"<str(<real>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ res1==r)

(∀: (String)s ∈ QSymbol ∧ t ∈ Term ∧
[[ATerm aterm=ATmake("<str(<placeholder>)>",s,t);
char *res0; ATerm *res1;
ATmatch(aterm,"<str(<placeholder>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t))

(∀: (String)s ∈ QSymbol ∧ (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("<str([<list>])>",s,l);
char *res0; ATermList res1;
ATmatch(aterm,"<str([<list>])>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(l,res1))

(∀: (String)s ∈ QSymbol ∧ (String)b ∈ Blob ∧
[[ATermBlob bl=ATmakeBlob(strlen(b),strdup(b));
ATerm aterm=ATmake("<str(<blob>)>",s,ATgetBlobSize(bl),ATgetBlobData(bl));
char *res0; int res1; char *res2;ATmatch(aterm,"<str(<blob>)>",&res0,&res1,&res2);]]:
strcmp(res0,s)==0 ∧ res1==strlen(b) ∧ strcmp(b,res2)==0)

(∀: (String)s ∈ QSymbol ∧ t ∈ Term ∧
[[ATerm aterm=ATmake("<str(<term>)>",s,t);
char *res0; ATerm *res1;
ATmatch(aterm,"<str(<term>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t))

(∀: (String)s ∈ QSymbol ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("<str(<str()>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<str(<str()>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (String)s ∈ QSymbol ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("<str(<appl()>)>",s,s2);
char *res0; char *res1;

ATmatch(aterm,"<str(<appl()>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (String)s ∈ QSymbol ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("<str(<str(2)>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<str(<str(2)>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (String)s ∈ QSymbol ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("<str(<appl(3)>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<str(<appl(3)>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (Int)i ∈ Int ∧
[[ATerm aterm=ATmake("[<int>]",i);
int res0;
ATmatch(aterm,"[<int>]",&res0);]]:
res0==i)

(∀: (Float)r ∈ Real ∧
[[ATerm aterm=ATmake("[<real>]",r);
double res0;
ATmatch(aterm,"[<real>]",&res0);]]:
res0==r)

(∀: p ∈ Term ∧
[[ATerm aterm=ATmake("[<placeholder>]",p);
ATerm res0;
ATmatch(aterm,"[<placeholder>]",&res0);]]:
ATisEqual(res0,p))

(∀: (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("[[<list>]]",l);
ATermList res0;
ATmatch(aterm,"[[<list>]]",&res0);]]:
ATisEqual(l,res0))

(∀: (String)b ∈ Blob ∧
[[ATermBlob bl=ATmakeBlob(strlen(b),strdup(b));
ATerm aterm=ATmake("[<blob>]",ATgetBlobSize(bl),ATgetBlobData(bl));
int res0; char *res1;ATmatch(aterm,"[<blob>]",&res0,&res1);]]:
res0==strlen(b) ∧ strcmp(b,res1)==0)

(∀: t ∈ Term ∧

[[ATerm aterm=ATmake("[<term>]",t);
ATerm *res0;
ATmatch(aterm,"[<term>]",&res0);]]:
ATisEqual(res0,t))

(∀: (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("[<str()>]",s2);
char *res0;
ATmatch(aterm,"[<str()>]",&res0);]]:
strcmp(res0,s2)==0)

(∀: (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("[<appl()>]",s2);
char *res0;
ATmatch(aterm,"[<appl()>]",&res0);]]:
strcmp(res0,s2)==0)

(∀: (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("[<str(2)>]",s2);
char *res0;
ATmatch(aterm,"[<str(2)>]",&res0);]]:
strcmp(res0,s2)==0)

(∀: (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("[<appl(3)>]",s2);
char *res0;
ATmatch(aterm,"[<appl(3)>]",&res0);]]:
strcmp(res0,s2)==0)

(∀: (String)s ∈ USymbol ∧ (Int)i ∈ Int ∧
[[ATerm aterm=ATmake("<appl(<int>)>",s,i);
char *res0; int res1;
ATmatch(aterm,"<appl(<int>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ res1==i)

(∀: (String)s ∈ USymbol ∧ (Float)r ∈ Real ∧
[[ATerm aterm=ATmake("<appl(<real>)>",s,r);
char *res0; double res1;
ATmatch(aterm,"<appl(<real>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ res1==r)

(∀: (String)s ∈ USymbol ∧ p ∈ Term ∧
[[ATerm aterm=ATmake("<appl(<placeholder>)>",s,p);
char *res0; ATerm res1;
ATmatch(aterm,"<appl(<placeholder>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,p))

(∀: (String)s ∈ USymbol ∧ (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("<appl([<list>])>",s,l);
char *res0; ATermList res1;
ATmatch(aterm,"<appl([<list>])>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(l,res1))

(∀: (String)s ∈ USymbol ∧ (String)b ∈ Blob ∧
[[ATermBlob bl=ATmakeBlob(strlen(b),strdup(b));
ATerm aterm=ATmake("<appl(<blob>)>",s,ATgetBlobSize(bl),ATgetBlobData(bl));
char *res0; int res1; char *res2;ATmatch(aterm,"<appl(<blob>)>",&res0,&res1,&res2);]]:
strcmp(res0,s)==0 ∧ res1==strlen(b) ∧ strcmp(b,res2)==0)

(∀: (String)s ∈ USymbol ∧ t ∈ Term ∧
[[ATerm aterm=ATmake("<appl(<term>)>",s,t);
char *res0; ATerm *res1;
ATmatch(aterm,"<appl(<term>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t))

(∀: (String)s ∈ USymbol ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("<appl(<str()>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<appl(<str()>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (String)s ∈ USymbol ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("<appl(<appl()>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<appl(<appl()>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)
(∀: (String)s ∈ USymbol ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("<appl(<str(2)>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<appl(<str(2)>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (String)s ∈ USymbol ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("<appl(<appl(3)>)>",s,s2);
char *res0; char *res1;
ATmatch(aterm,"<appl(<appl(3)>)>",&res0,&res1);]]:
strcmp(res0,s)==0 ∧ strcmp(res1,s2)==0)

(∀: (String)s ∈ USymbol ∧ (Float)f ∈ Real ∧ p ∈ Term ∧
[[ATerm aterm=ATmake("<appl(<real>,<placeholder>)>",s,f,p);
char *res0; double res1; ATerm res2;
ATmatch(aterm,"<appl(<real>,<placeholder>)>",&res0,&res1,&res2);]]:

strcmp(res0,s)==0 ∧ res1==f ∧ ATisEqual(res2,p))

(∀: (String)s ∈ USymbol ∧ l ∈ List ∧ (String)b ∈ Blob ∧ (String)s2 ∈ QSymbol ∧
(Float)f ∈ Real ∧ [[ATerm aterm=ATmake("<appl(<term>,<blob>,<str(1)>,<real>)>",s,l,strlen(b),strdup(b),s2,f);
char *res0; ATerm res1; int res2; char *res3; char *res4; double res5;
ATmatch(aterm,"<appl(<term>,<blob>,<str(1)>,<real>)>",&res0,&res1,&res2,&res3,&res4,&res5);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,l) ∧ res2==strlen(b) ∧ strcmp(res3,b)==0 ∧ strcmp(res4,s2)==0 ∧
res5==f)

(∀: (String)s ∈ USymbol ∧ p ∈ Term ∧ (String)s2 ∈ USymbol ∧ (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("<appl(<placeholder>,<appl(23)>,<list>)>",s,p,s2,l);
char *res0; ATerm res1; char *res2; ATermList res3;
ATmatch(aterm,"<appl(<placeholder>,<appl(23)>,<list>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,p) ∧ strcmp(res2,s2)==0 ∧ ATisEqual(res3,ATremoveAnnotations(l)))

(∀: (String)s ∈ USymbol ∧ (String)b ∈ Blob ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("<appl(<blob>,<appl(42)>)>",s,strlen(b),strdup(b),s2);
char *res0; int res1; char *res2; char *res3;
ATmatch(aterm,"<appl(<blob>,<appl(42)>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ res1==strlen(b) ∧ strcmp(res2,b)==0 ∧ strcmp(res3,s2)==0)

(∀: (String)s ∈ USymbol ∧ (Int)i ∈ Int ∧ (Int)i2 ∈ Int ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("<appl(<int>,<int>,<str(1,2,3)>)>",s,i,i2,s2);
char *res0; int res1; int res2; char *res3;
ATmatch(aterm,"<appl(<int>,<int>,<str(1,2,3)>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ res1==i ∧ res2==i2 ∧ strcmp(res3,s2)==0)

(∀: (String)s ∈ USymbol ∧ t ∈ Term ∧ (String)b ∈ Blob ∧ (String)b2 ∈ Blob ∧
[[ATerm aterm=ATmake("<appl(<term>,<blob>,<blob>)>",s,t,strlen(b),strdup(b),strlen(b2),strdup(b2));
char *res0; ATerm res1; int res2; char *res3; int res4; char *res5;
ATmatch(aterm,"<appl(<term>,<blob>,<blob>)>",&res0,&res1,&res2,&res3,&res4,&res5);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t) ∧ res2==strlen(b) ∧ strcmp(res3,b)==0 ∧ res4==strlen(b2) ∧
strcmp(res5,b2)==0)

(∀: (String)s ∈ USymbol ∧ (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)i3 ∈ Int ∧
[[ATerm aterm=ATmake("<appl(<int>,<int>,<int>)>",s,i1,i2,i3);
char *res0; int res1; int res2; int res3;
ATmatch(aterm,"<appl(<int>,<int>,<int>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ res1==i1 ∧ res2==i2 ∧ res3==i3)

(∀: (String)s ∈ USymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧
[[ATerm aterm=ATmake("<appl(<term>,<term>,<term>)>",s,t1,t2,t3);
char *res0; ATerm res1; ATerm res2; ATerm res3;
ATmatch(aterm,"<appl(<term>,<term>,<term>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t1) ∧ ATisEqual(res2,t2) ∧ ATisEqual(res3,t3))

(∀: (String)s ∈ QSymbol ∧ (Float)f ∈ Real ∧ p ∈ Term ∧
[[ATerm aterm=ATmake("<str(<real>,<placeholder>)>",s,f,p);
char *res0; double res1; ATerm res2;
ATmatch(aterm,"<str(<real>,<placeholder>)>",&res0,&res1,&res2);]]:
strcmp(res0,s)==0 ∧ res1==f ∧ ATisEqual(res2,p))

(∀: (String)s ∈ QSymbol ∧ l ∈ List ∧ (String)b ∈ Blob ∧ (String)s2 ∈ QSymmbol ∧ (Float)f ∈ Real ∧
[[ATerm aterm=ATmake("<str(<term>,<blob>,<str(1)>,<real>)>",s,l,strlen(b),strdup(b),s2,f);
char *res0; ATerm res1; int res2; char *res3; char *res4; double res5;
ATmatch(aterm,"<str(<term>,<blob>,<str(1)>,<real>)>",&res0,&res1,&res2,&res3,&res4,&res5);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,l) ∧ res2==strlen(b) ∧ strcmp(res3,b)==0 ∧ strcmp(res4,s2)==0 ∧ res5==f)

(∀: (String)s ∈ QSymbol ∧ p ∈ Term ∧ (String)s2 ∈ USymbol ∧ (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("<str(<placeholder>,<appl(23)>,<list>)>",s,p,s2,l);
char *res0; ATerm res1; char *res2; ATermList res3;
ATmatch(aterm,"<str(<placeholder>,<appl(23)>,<list>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,p) ∧ strcmp(res2,s2)==0 ∧ ATisEqual(res3,ATremoveAnnotations(l)))

(∀: (String)s ∈ QSymbol ∧ (String)b ∈ Blob ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("<str(<blob>,<appl(42)>)>",s,strlen(b),strdup(b),s2);
char *res0; int res1; char *res2; char *res3;
ATmatch(aterm,"<str(<blob>,<appl(42)>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ res1==strlen(b) ∧ strcmp(res2,b)==0 ∧ strcmp(res3,s2)==0)

(∀: (String)s ∈ QSymbol ∧ (Int)i ∈ Int ∧ (Int)i2 ∈ Int ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("<str(<int>,<int>,<str(1,2,3)>)>",s,i,i2,s2);
char *res0; int res1; int res2; char *res3;
ATmatch(aterm,"<str(<int>,<int>,<str(1,2,3)>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ res1==i ∧ res2==i2 ∧ strcmp(res3,s2)==0)

(∀: (String)s ∈ QSymbol ∧ t ∈ Term ∧ (String)b ∈ Blob ∧ (String)b2 ∈ Blob ∧
[[ATerm aterm=ATmake("<str(<term>,<blob>,<blob>)>",s,t,strlen(b),strdup(b),strlen(b2),strdup(b2));
char *res0; ATerm res1; int res2; char *res3; int res4; char *res5;
ATmatch(aterm,"<str(<term>,<blob>,<blob>)>",&res0,&res1,&res2,&res3,&res4,&res5);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t) ∧ res2==strlen(b) ∧ strcmp(res3,b)==0 ∧ res4==strlen(b2) ∧
strcmp(res5,b2)==0)

(∀: (String)s ∈ QSymbol ∧ (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)i3 ∈ Int ∧
[[ATerm aterm=ATmake("<str(<int>,<int>,<int>)>",s,i1,i2,i3);
char *res0; int res1; int res2; int res3;
ATmatch(aterm,"<str(<int>,<int>,<int>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ res1==i1 ∧ res2==i2 ∧ res3==i3)

(∀: (String)s ∈ QSymbol ∧ t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧
[[ATerm aterm=ATmake("<str(<term>,<term>,<term>)>",s,t1,t2,t3);
char *res0; ATerm res1; ATerm res2; ATerm res3;
ATmatch(aterm,"<str(<term>,<term>,<term>)>",&res0,&res1,&res2,&res3);]]:
strcmp(res0,s)==0 ∧ ATisEqual(res1,t1) ∧ ATisEqual(res2,t2) ∧ ATisEqual(res3,t3))

(∀: (Float)f ∈ Real ∧ p ∈ Term ∧
[[ATerm aterm=ATmake("[<real>,<placeholder>]",f,p);
double res1; ATerm res2;
ATmatch(aterm,"[<real>,<placeholder>]",&res1,&res2);]]:
res1==f ∧ ATisEqual(res2,p))

(∀: l ∈ List ∧ (String)b ∈ Blob ∧ (String)s2 ∈ QSymmbol ∧ (Float)f ∈ Real ∧
[[ATerm aterm=ATmake("[<term>,<blob>,<str(1)>,<real>]",l,strlen(b),strdup(b),s2,f);
ATerm res1; int res2; char *res3; char *res4; double res5;
ATmatch(aterm,"[<term>,<blob>,<str(1)>,<real>]",&res1,&res2,&res3,&res4,&res5);]]:
ATisEqual(res1,l) ∧ res2==strlen(b) ∧ strcmp(res3,b)==0 ∧ strcmp(res4,s2)==0 ∧ res5==f)

(∀: p ∈ Term ∧ (String)s2 ∈ USymbol ∧ (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("[<placeholder>,<appl(23)>,<list>]",p,s2,l);
ATerm res1; char *res2; ATermList res3;
ATmatch(aterm,"[<placeholder>,<appl(23)>,<list>]",&res1,&res2,&res3);]]:
ATisEqual(res1,p) ∧ strcmp(res2,s2)==0 ∧ ATisEqual(res3,l))

(∀: (String)b ∈ Blob ∧ (String)s2 ∈ USymbol ∧
[[ATerm aterm=ATmake("[<blob>,<appl(42)>]",strlen(b),strdup(b),s2);
int res1; char *res2; char *res3;
ATmatch(aterm,"[<blob>,<appl(42)>]",&res1,&res2,&res3);]]:
res1==strlen(b) ∧ strcmp(res2,b)==0 ∧ strcmp(res3,s2)==0)

(∀: (Int)i ∈ Int ∧ (Int)i2 ∈ Int ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("[<int>,<int>,<str(1,2,3)>]",i,i2,s2);
int res1; int res2; char *res3;
ATmatch(aterm,"[<int>,<int>,<str(1,2,3)>]",&res1,&res2,&res3);]]:
res1==i ∧ res2==i2 ∧ strcmp(res3,s2)==0)

(∀: t ∈ Term ∧ (String)b ∈ Blob ∧ (String)b2 ∈ Blob ∧
[[ATerm aterm=ATmake("[<term>,<blob>,<blob>]",t,strlen(b),strdup(b),strlen(b2),strdup(b2));
ATerm res1; int res2; char *res3; int res4; char *res5;
ATmatch(aterm,"[<term>,<blob>,<blob>]",&res1,&res2,&res3,&res4,&res5);]]:
ATisEqual(res1,t) ∧ res2==strlen(b) ∧ strcmp(res3,b)==0 ∧ res4==strlen(b2) ∧ strcmp(res5,b2)==0)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)i3 ∈ Int ∧
[[ATerm aterm=ATmake("[<int>,<int>,<int>]",i1,i2,i3);
int res1; int res2; int res3;
ATmatch(aterm,"[<int>,<int>,<int>]",&res1,&res2,&res3);]]:

res1==i1 ∧ res2==i2 ∧ res3==i3)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ t3 ∈ Term ∧
[[ATerm aterm=ATmake("[<term>,<term>,<term>]",t1,t2,t3);
ATerm res1; ATerm res2; ATerm res3;
ATmatch(aterm,"[<term>,<term>,<term>]",&res1,&res2,&res3);]]:
ATisEqual(res1,t1) ∧ ATisEqual(res2,t2) ∧ ATisEqual(res3,t3))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)i3 ∈ Int ∧
[[ATerm
aterm=ATmake("[<int>,<int>,<int>,<int>,<int>,<int>,
<int>,<int>,<int>,<int>,<int>,<int>]",i1,i2,i3,i1,i2,i3,i1,i2,i3,i1,i2,i3);
int res1; int res2; int res3; int res4; int res5; int res6; int res7; int res8; int res9; int res10; int res11; int res12;
ATmatch(aterm,"[<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,
<int>,<int>,<int>]",&res1,&res2,&res3,&res4,&res5,&res6,&res7,
&res8,&res9,&res10,&res11,&res12);]]:
res1==i1 ∧ res2==i2 ∧ res3==i3 ∧ res4==i1 ∧ res5==i2 ∧ res6==i3 ∧ res7==i1 ∧ res8==i2 ∧
res9==i3 ∧ res10==i1 ∧ res11==i2 ∧ res12==i3)

(∀: (Int)i ∈ Int ∧
[[ATerm aterm=ATmake("appl(<int>,5)",i);int res1;
ATmatch(aterm,"appl(<int>,5)",&res1);]]:
res1==i)

(∀: t ∈ Term ∧
[[ATerm aterm=ATmake("appl(<term>,5)",t);ATerm res1;
ATmatch(aterm,"appl(<term>,5)",&res1);]]:
ATisEqual(res1,t))

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ (Int)i3 ∈ Int ∧
[[ATerm
aterm=ATmake("appl(<int>,<int>,<int>,<int>,<int>,<int>,<int>,
<int>,<int>,<int>,<int>,<int>)",i1,i2,i3,i1,i2,i3,i1,i2,i3,i1,i2,i3);
int res1; int res2; int res3; int res4; int res5; int res6; int res7; int res8; int res9; int res10; int res11; int res12;
ATmatch(aterm,"appl(<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>,<int>)
",&res1,&res2,&res3,&res4,&res5,&res6,&res7,&res8,&res9,&res10,&res11,&res12);]]:
res1==i1 ∧ res2==i2 ∧ res3==i3 ∧ res4==i1 ∧ res5==i2 ∧ res6==i3 ∧ res7==i1
∧ res8==i2 ∧ res9==i3 ∧ res10==i1 ∧ res11==i2 ∧ res12==i3)

(∀: p ∈ Term ∧ (String)s2 ∈ USymbol ∧ (ATermList)l ∈ List ∧
[[ATerm aterm=ATmake("[<placeholder>,<appl(23)>,<list>][1,3]",p,s2,l);
ATerm res1; char *res2; ATermList res3;
ATmatch(aterm,"[<placeholder>,<appl(23)>,<list>][1,3]",&res1,&res2,&res3);]]:
ATisEqual(res1,p) ∧ strcmp(res2,s2)==0 ∧ ATisEqual(res3,l))

(∀: (String)s ∈ Qsymbol ∧ (String)b ∈ Blob ∧ (String)s2 ∈ USymbol ∧

[[ATerm aterm=ATmake("<appl(<blob>,<appl(42)>)[4,4],[5,5]>",s,strlen(b),strdup(b),s2);
char *res0; int res1; char *res2; char *res3;
ATmatch(aterm,"<appl(<blob>,<appl(42)>)[4,4],[5,5]",&res0,&res1,&res2,&res3);]]:
res1==strlen(b) ∧ strcmp(res2,b)==0 ∧ strcmp(res3,s2)==0)

(∀: (Int)i ∈ Int ∧ (Int)i2 ∈ Int ∧ (String)s2 ∈ QSymbol ∧
[[ATerm aterm=ATmake("appl(<int>,<int>,<str(1,2,3)>)[1,1],[2,2],[3,3]",i,i2,s2);
int res1; int res2; char *res3;
ATmatch(aterm,"appl(<int>,<int>,<str(1,2,3)>)[1,1],[2,2],[3,3]",&res1,&res2,&res3);]]:
res1==i ∧ res2==i2 ∧ strcmp(res3,s2)==0)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ [[ATerm diffs; ATerm template; ATbool b=ATdiff((ATmakeInt(i1)),(ATmakeInt(i2)),
&template, &diffs);ATbool c= ((i1==i2) == ATisEqual(diffs,ATparse("[]")));]]: c)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm diffs; ATerm template; ATbool b=ATdiff(t1,t2, &template, &diffs);
ATbool c= (ATisEqual(ATremoveAnnotations(t1),ATremoveAnnotations(t2)) == ATisEqual(diffs,ATparse("[]")));]]:
c)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm diffs; ATerm template; ATerm appl1=ATmake("f(<term>)",t1);
ATerm
appl2=ATmake("f(<term>)",t2); ATbool b=ATdiff(appl1,appl2, &template, &diffs);ATbool c=
(ATisEqual(ATremoveAnnotations(t1),ATremoveAnnotations(t2)) == ATisEqual(template,ATmake("f(<term>)",t1)));]]:
c)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm diffs; ATerm template; ATerm appl1=ATmake("[<term>]",t1);
ATerm
appl2=ATmake("[<term>]",t2);
ATbool b=ATdiff(appl1,appl2, &template, &diffs);
ATbool c= (ATisEqual(ATremoveAnnotations(t1),ATremoveAnnotations(t2)) == ATisEqual(template,
ATmake("[<term>]",t1)));]]: c)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[ATerm diffs; ATerm template;
ATerm appl1=ATmake("[3.14,<term>,2]",t1); ATerm
appl2=ATmake("[3.14,<term>,2]",t2);
ATbool b=ATdiff(appl1,appl2, &template, &diffs);
ATbool c= (ATisEqual(ATremoveAnnotations(t1),ATremoveAnnotations(t2)) ==
ATisEqual(template,ATmake("[3.14,<term>,2]",t1)));]]: c)

(∀: (Int)i ∈ Int: ATcompare(ATmake("f(<int>)",i),ATmake("<int>",i))<0)

(∀: (Int)i ∈ Int ∧ (Float)f ∈ Real: ATcompare(ATmake("<int>",i),ATmake("<real>",f))<0)

(∀: (Float)f ∈ Real ∧ (ATermList)l ∈ List: ATcompare(ATmake("<real>",f),l)<0)

(∀: (ATermList)l ∈ List ∧ (String)s ∈ Placeholder: ATcompare(l,ATmakePlaceholder(s))<0)

(∀: (String)s ∈ Placeholder ∧ (String)b ∈ Blob: ATcompare(ATmakePlaceholder(s),ATmakeBlob(strlen(b),strdup(b))<0))

(∀: t ∈ Term: ATcompare(t,t)==0)

(∀: (String)s1 ∈ USymbol ∧ (String)s2 ∈ USymbol ∧ [[ATerm appl1=ATmake("<appl(1)>",s1);ATerm appl2=ATmake("<appl(1)>",s2);]]: strcmp(s1,s2)==ATcompare(appl1,appl2))

(∀: (String)s ∈ USymbol ∧ t ∈ Term ∧ [[
ATerm atappl = ATmake("<appl()>",s);
ATerm atappl1 = ATmake("<appl(<term>)>",s,t);
ATerm atappl2 = ATmake("<appl(<term>,<term>)>",s,t,t);]]:
ATcompare(atappl,atappl1)<0 ∧ ATcompare(atappl,atappl2)<0 ∧ ATcompare(atappl1,atappl2)<0)

(∀: t ∈ Term ∧ [[ATerm atlist = ATmake("[]");
ATerm atlist1 = ATmake("[<term>]",t);
ATerm atlist2 = ATmake("[<term>,<term>]",t,t);]]:
ATcompare(atlist,atlist1)<0 ∧ ATcompare(atlist,atlist2)<0 ∧ ATcompare(atlist1,atlist2)<0)

(∀: t0 ∈ Term ∧ [[ATerm t=ATremoveAnnotations(t0);int i=6666; ATerm t2=ATsetAnnotation(t,ATmake("<int>",i),ATmake("<int>",i));
ATerm t3=ATsetAnnotation(t2,ATmake("<int>",i+1),ATmake("<int>",i+1));]]:
ATcompare(t,t2)<0 ∧ ATcompare(t2,t3)<0 ∧ ATcompare(t,t3)<0)

(∀: (String)s1 ∈ Blob ∧ (String)s2 ∈ Blob ∧ [[int c=strcmp(s1,s2);]]:
ATcompare(ATmakeBlob(strlen(s1),strdup(s1)),ATmakeBlob(strlen(s2),strdup(s2)))==c)

(∀: (Int)i1 ∈ Int ∧ (Int)i2 ∈ Int ∧ [[ATbool c=(i1==i2); ATbool d=ATcompare(ATmakeInt(i1),ATmakeInt(i2))==0;]]:
c==d)

(∀: (Float)f1 ∈ Real ∧ (Float)f2 ∈ Real ∧ [[ATbool c=(f1==f2);ATbool d=ATcompare(ATmakeReal(f1),ATmakeReal(f2))==0;]]:
c==d)

(∀: (String)s1 ∈ Placeholder ∧ (String)s2 ∈ Placeholder ∧ [[int c=strcmp(s1,s2);]]:
ATcompare(ATmakePlaceholder(s1),ATmakePlaceholder(s2))==c)

(∀: t ∈ Term: AT_calcCoreSize(t)>0)

(∀: (Int)size ∈ Size ∧ t ∈ Term ∧ [[ATermList atlist=ATempty; int i; for(i=0;i<size;i++) atlist=ATinsert(atlist,t);]]
:AT_calcSubterms(atlist)==size*(1+AT_calcSubterms(t)) ∧ ATcalcUniqueSubterms(atlist)-size-1 == ATcalcUniqueSubterms(t) ∧)

(∀: (ATermList)l ∈ List ∧ t ∈ Term ∧ [[ATermList atl=l; ATerm att=t; ATerm result = ATmake("f(<term>,<term>)",atl,att);AT_markTerm(result);]]:

IS_MARKED(atl->header) ∧ IS_MARKED(att->header) ∧ IS_MARKED(result->header) ∧
[[AT_unmarkTerm(result);]] ∧ !IS_MARKED(atl->header) ∧ !IS_MARKED(att->header))

(∀: 2≤i<10 ∧
[[static ATerm ts1 = NULL, ts2 = NULL; ATerm ts[100]; ATprotect(&ts1); ts1 = ATmake("unique-1"); int
j; for(j=0;j<i;j++)
ts[j]=ATmake("f(<int>)",j); ; AT_collect(2);]]:
AT_isValidTerm(ts1) ∧
[[ATunprotect(&ts1); AT_collect(2); ts[0] = (ATerm)"garbage";]] ∧
!AT_isValidTerm(ts[0]) ∧
AT_isValidTerm(ts[1]) ∧
AT_isInsideValidTerm(ts[0]) == NULL ∧
AT_isInsideValidTerm((ATerm)(((char *)ts[1])+2)) ≠ NULL ∧
[[ATprotectMemory((void *)ts, sizeof(ATerm)*2);AT_collect(2);]] ∧
AT_isValidTerm(ts[1]) ∧ [[ATunprotectMemory((void *)ts);AT_collect(2);]])

(∀: t ∈ Term ∧
t2 ∈ Term ∧
[[static ATerm ts1 = NULL, ts2 = NULL; static ATerm ts[2]; ATprotect(&ts1); ts1 = t; ts2 = t2; AT_collect(2);]]:
AT_isValidTerm(ts1) ∧
[[ATunprotect(&ts1);AT_collect(2); ts[0] = (ATerm)"garbage"; ts[1] = ATmake("unique-3");]] ∧
!AT_isValidTerm(ts[0]) ∧
AT_isValidTerm(ts[1]) ∧
AT_isInsideValidTerm(ts[0]) == NULL ∧
AT_isInsideValidTerm((ATerm)(((char *)ts[1])+4)) ≠ NULL ∧
[[ATprotectMemory((void *)ts, sizeof(ATerm)*2); AT_collect(2);]] ∧
AT_isValidTerm(ts[1]) ∧
[[ATunprotectMemory((void *)ts); AT_collect(2);]])

(∀: 0≤aa<1: [[AT_validateFreeList(100);]] ∧ (∀: t ∈ Term ∧
[[ATerm aterm = t;]] ∧ 0≤i<10 : [[aterm = ATmake("succ(<int>,<term>)", i, aterm);]]
∧ !AT_inAnyFreeList(aterm)) ∧ AT_getAllocatedCount()>0 ∧
[[FILE *file; file = fopen("file3.txt","w");]] ∧ [[AT_printAllTerms(file);AT_printAllAFunCounts(file)]]
∧ [[fclose(file);]] ∧ [[AT_validateFreeList(10)]] ∧ AT_calcAllocatedSize()>0
)

(∀: t ∈ Term ∧ [[ATerm aterm = t;]] ∧ 0≤i<1000000 : [[aterm = ATmake("succ(<int>,<term>)", i,
aterm);]]
∧ aterm)

(∀: t ∈ Term ∧ [[AT_statistics()]]: 1==1)

(∀: 0≤i<1: 1==1 ∧ [[ATsetChecking(ATfalse)]] ∧ ATgetChecking()==0 ∧ [[ATsetChecking(ATtrue)]] ∧
ATgetChecking()==1)

(∀: 0≤i<1 ∧ [[int k; int l;AT_getBafVersion(&k,&l);]]: 1==1)

(∀: (String)t ∈ AbortTerm ∧ [[ATbool parse_error_encountered = ATfalse; void abort_handler(const char *format, va_list
args) parse_error_encountered = ATtrue;
("probleem!!");;ATsetAbortHandler(abort_handler); ATerm aterm=ATparse(t);ATsetAbortHandler(NULL);]]: parse_error_encountered)

(∀: t ∈ Term ∧ [[void abort_handler(const char *format, va_list args)
ATerm aterm=va_arg(args,ATerm); FILE *file; file = fopen("file.txt","w"); ATfprintf(file,"%t",aterm); fclose(file);
;
ATsetAbortHandler(abort_handler); ATabort("tada",t);FILE *file2; file2 = fopen("file.txt", "rb"); ATerm t2=ATreadFromTextFile(file2);fclose(file2);
ATbool b=ATfalse; if (ATgetType(t)==AT_BLOB ∧ ATgetType(t2)==AT_BLOB) if (strcmp(ATgetBlobData((ATermBlob)t),ATgetBlobData((ATermBlob)t2))==0) b=ATtrue; ;
]]: ATisEqual(t,t2) ∨ b)

(∀: t ∈ Term ∧ [[void error_handler(const char *format, va_list args)
ATerm aterm=va_arg(args,ATerm); FILE *file; file = fopen("file.txt","w"); ATfprintf(file,"%t",aterm); fclose(file);
;
ATsetErrorHandler(error_handler); ATerror("tada",t); FILE *file2; file2 = fopen("file.txt", "rb"); ATerm t2=ATreadFromTextFile(file2);fclose(file2);
ATbool b=ATfalse; if (ATgetType(t)==AT_BLOB ∧ ATgetType(t2)==AT_BLOB) if (strcmp(ATgetBlobData((ATermBlob)t),ATgetBlobData((ATermBlob)t2))==0) b=ATtrue; ;
]]: ATisEqual(t,t2) ∨ b)

(∀: t ∈ Term ∧ [[void warning_handler(const char *format, va_list args)
ATerm aterm=va_arg(args,ATerm); FILE *file; file = fopen("file.txt","w"); ATfprintf(file,"%t",aterm); fclose(file);
;
ATsetWarningHandler(warning_handler); ATwarning("tada",t); FILE *file2; file2 = fopen("file.txt", "rb");
ATerm t2=ATreadFromTextFile(file2);
fclose(file2);
ATbool b=ATfalse; if (ATgetType(t)==AT_BLOB ∧ ATgetType(t2)==AT_BLOB) if (strcmp(ATgetBlobData((ATermBlob)t),ATgetBlobData((ATermBlob)t2))==0) b=ATtrue; ;
]]: ATisEqual(t,t2) ∨ b)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[int c=1456;char *tekst="tekst";
ATprintf("%t",t1);
ATprintf("a %c b %d c %i d %o e %u f %x g %X %t",'a',42,43,610,7235,c,c,t1);
ATprintf("%t h %e i %E j %f k %g l %G m %p n %s",t1,663.926, 6663.926, 5.5, 666.666, 666.666, &c,tekst);
ATprintf("a %c b %d c %i d %o e %u f %x g %X %t h %e i %E j %f k %g l %G m %p n %s",'a',42,43,610,7235,c,c,t1,663.926, 6663.926, 5.5, 666.666, 666.666, &c,tekst);
ATprintf("a %c b %d c %i d %o e %u f %x g %X %t %t h %e i %E j %f k %g l %G m %p n %s",'a',42,43,610,7235,c,c,t1,t2,663.926, 6663.926, 5.5, 666.666, 666.666, &c,tekst);]]: 1==1)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[char line1[1000];char line2[1000]; FILE *file1; FILE *file2;

file1 = fopen("file1.txt","w"); ATfprintf(file1,"%n",t1); fclose(file1);
file1 = fopen("file1.txt", "rb"); int i=fgets(line1, sizeof(line1), file1); fclose(file1);
file2 = fopen("file2.txt","w"); ATfprintf(file2,"%n",t2); fclose(file2);
file2 = fopen("file2.txt", "rb"); int j=fgets(line2, sizeof(line2), file2);fclose(file2);
ATbool b1=ATisEqual(t1,t2);
ATbool b2=strcmp(line1,line2)==0;
]]: !b1 ∨ b2)

(∀: t1 ∈ Term ∧ t2 ∈ Term ∧ [[char line1[1000];char line2[1000]; FILE *file1; FILE *file2;
file1 = fopen("file1.txt","w"); ATfprintf(file1,"%h",t1); fclose(file1);
file1 = fopen("file1.txt", "rb"); int i=fgets(line1, sizeof(line1), file1); fclose(file1);
file2 = fopen("file2.txt","w"); ATfprintf(file2,"%h",t2); fclose(file2);
file2 = fopen("file2.txt", "rb"); int j=fgets(line2, sizeof(line2), file2);fclose(file2);
ATbool b1=(ATisEqual(t1,t2));
ATbool b2=(strcmp(line1,line2)==0);
]]: b1 == b2)