ADATE User Manual

Østfold University College

Geir Vattekar

March 2006

1. Introduction

ADATE (Automatic Design of Algorithms Through Evolution) is a system for automatic programming developed by Roland Olsson at Østfold University College. Automatic programming is inductive inference of algorithms. The main focus of this manual is use of the system. By use we mean the procedure of setting up the ADATE run-time environment, implementing problems, compiling and running ADATE and the examination of the output from ADATE.

The manual also gives as a short introduction to the background and inner workings of the system. In addition to describing the technicalities of using ADATE it is intended to give some practical advice and help in real problem implementation issues.

The rest of this introduction describes the objectives of the manual, points out its limitations and explains which groups of users the manual targets.

1.1 Purpose and objectives

The main reason for writing the manual is that we wish to introduce ADATE to a wider audience than it has today and that existing documentation was outdated and insufficient. We hope that the manual will make ADATE more useful to the machine learning and evolutionary computation (EC) communities. Also, the manual will serve as help for the students in the intelligent systems programme at HiØ. The manual has three main objectives:

- Make it possible to learn how to use the system without having to refer to anything but the manual.
- Aid in problem implementation
- Give a general introduction to the system.

1.2 Who the manual is for

Specifically there are two main groups of users the manual targets. We try in particular to accommodate machine learning professionals and students in graduate programmes in machine learning interested in:

- General knowledge about ADATE as a machine learning and an EC system
- Finding a system to solve a specific problem
- Comparing ADATE to other EC systems with regard to use and abilities.
- Using ADATE to learn automatic programming

We recommend that the user is somewhat familiar with the following to read the manual:

- Machine learning terminology
- Linux
- The ML programming language
- MPI if ADATE is to run on a Linux cluster

If the reader is not familiar with some or all of this we refer to sources in the appendices of the manual that can aid in acquiring this knowledge.

1.3 Limitations and restrictions

The manual is not intended as a reference for experienced ADATE users. The manual does not try to teach everything about the ADATE system but merely to provide a foundation for users to build on. The manual does, however, contain references to more detailed information in its appendices.

The manual does not serve as a technical manual for the ADATE system, so the internal workings of the system are not treated in detail. However, it is necessary to briefly describe some evolutionary computation aspects of ADATE . Readers may need this if confused about the contents of the specification and output files.

2. The ADATE system

This chapter will give an introduction to the ADATE system. We do not describe the internal workings of the system with a high level of detail, as this would be beyond the scope of this text. We try to limit the discussion to the information that is necessary to properly be able to use the system, such as implementing problems and understanding its output, and place ADATE in the landscape of machine learning techniques.

Programming may be seen as a process that starts with a specification and generates a program that satisfies it. Automatic programming is then the automation of this process. Automatic programming is a sub-field of machine learning and can itself be divided into fields like inductive logical programming and EC.

ADATE belongs to the field of EC with other techniques such as genetic algorithms, evolution strategies and genetic programming. Common for EC systems is that they employ search techniques that are inspired by basic biological principles of evolution, like for instance mutation and crossover. We will see in this chapter that ADATE draws on this analogy with nature both in terminology and functionality.

2.1 The ADATE search

The search for programs conducted by ADATE is global and mostly systematic, that is, without randomization. A series of heuristics is utilized to increase the efficiency of the search. The rest of this section will look at the main search strategy of ADATE.

ADATE builds a collection of programs during the search called the kingdom in analogy with a kingdom in biological taxonomy. Programs are viewed as individuals.. The kingdom consists initially of only one individual, which usually is the empty program, that is, a program that only raises an exception. Adding new individuals then expands the kingdom. New individuals are generated as ADATE tries to transform (change) programs in the kingdom by applying so-called transformations to them. Transformations are described later in this chapter.

If ADATE finds an individual in this manner that it deems better than programs already in the kingdom it is inserted into the kingdom. It may also replace programs that are considered worse. ADATE uses an evaluation value to assess how good individuals are. How this value is found is described later in this chapter.

By continuing to produce and insert individuals in this manner, ADATE will find better and better programs as the search progresses. As ADATEs transforms its programs using methods that are inspired by biological reproduction, ADATEs search can to some extent be seen as artificial evolution. In the following sections, parts of this search will be examined in more detail.

2.2 Kingdom Management

The collection of programs (the kingdom) that ADATE builds and maintains during a run is described in this section. As mentioned, the kingdom starts with a single individual and then expands as the evolution progresses. It is the structure and management of the kingdom that will be the focus of this section.

2.2.1 Kingdom structure

ADATE classifies its individuals into a hierarchical structure. By analogy with biology, this structure is based on the scientific classification used in the biological sciences to group organisms. Classification is based on Linnean Taxonomy, which is the system used by ADATE. Linnean Taxonomy groups the organisms after their physical characteristics. The groupings in the hierarchy are called taxa (singular: taxon).

In the following table, one can see the groupings in the Linnean Taxonomy from the most general (kingdom) to the most specific (species) and the corresponding type of ADATE individuals that will constitute the taxa.

Biologic taxon	Usage inADATE
Kingdom	All Individuals
Phylum	These groupings are used for parts of the ADATE system that are not treated in this manual.
Class	
Order	
Family	Consists of a base individual, an embedding genus and an output genus.
Genus	A collection of either embedding or output individuals
Species	Individuals generated from one “founding individual” through so-called compound transformations.

- An individual is the lowest classification and represents a program
- A base individual is the smallest one found so far for a given syntactic size and execution time limit.
- An output individual is an individual that has the same evaluation value but is semantically different from the base individual and all other output individuals of the family.
- An embedding individual has help function definitions with extra arguments compared with the help functions defined in the base individual (see section 2.3)
- A species is represented by a single individual from which it is generated by so-called compound transformations (see section 2.3)

2.2.2 Size-time-complexity grid

The individuals that ADATE keeps in the kingdom are placed by their syntactic complexity and time complexity into grids. The complexity measures that are used are built into ADATE. They will not be treated in detail but time complexity can be seen as the time an individual uses to run the training examples and the syntactic complexity as the size of the program in bits.

Each square in the grid consists of a family with at least one base individual genus and possibly other genera as well, each of which may contain several species. Note that the individuals in the grid are only representatives of their species. At the start of the induction, the initial individual is used in all grid squares. An illustration of a grid is given below.

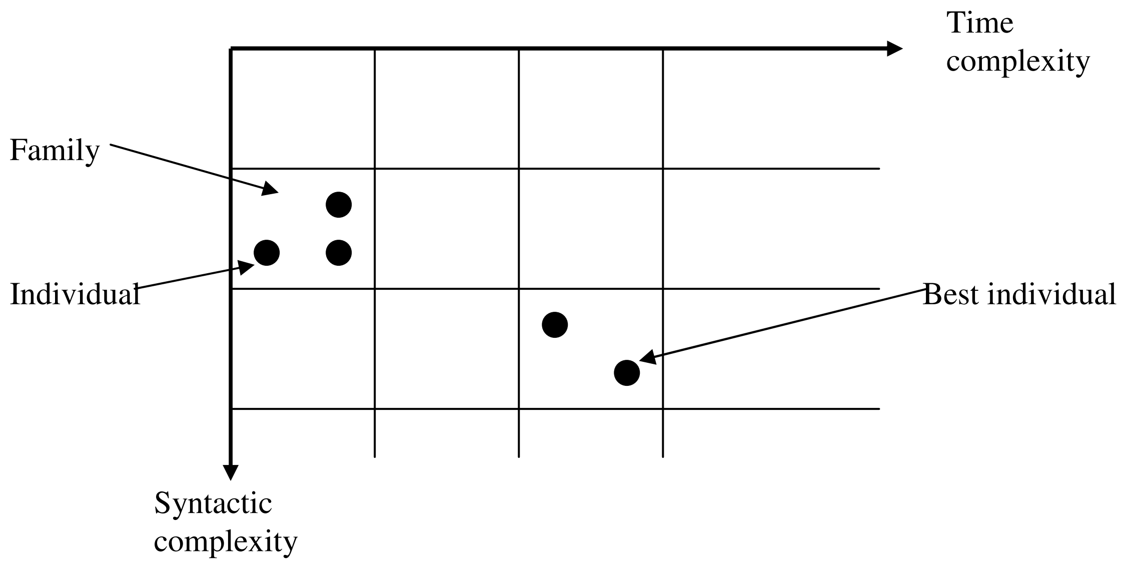


Fig 1: Size-time-complexity grid. Each square represents a span of time -and syntactic complexity that the family of the square fits in. The complexity is increasing along the axes. The best-evaluated individual will be located to the outmost right and down in the grid (see next section)

2.2.3 Insertion and evaluation

ADATE compares a newly generated individual with the individuals in the family it belongs to and possibly replaces them. It will replace base individuals that have a worse evaluation value through a downwards and rightwards flood-fill. Note that this will leave the individual with the best evaluation value outermost right and down in the grid.

ADATE uses three different program evaluation functions to calculate the evaluation value of individuals and three different syntactic complexity measures, all of which measure program size in bits. One time-complexity grid is used for all combinations of program evaluation functions and complexity measures and that makes a total of nine grids that make up the kingdom.

The user supplies one output evaluation function that is used when calculating the three program evaluation functions which are internal to ADATE. Without explaining each function in detail it suffices to say that they all calculate an evaluation value based on the number of correct and wrong training examples, the number of memory and time limit overflows and a user defined grade.

2.3 Transformations

Transformations are the search operators that are used to alter the individuals in the kingdom during reproduction. Any individual can be backtracked through a series of such transformations to the initial program. ADATE has six basic forms of transformations, called atomic transformations:

1. R (Replacement): Replacement changes part of the individual with new expressions.
2. REQ (Replacement without making the individuals evaluation value worse): Does the same as Replacement but now the new individual is guaranteed to have an equal or better evaluation value.
3. ABSTR (Abstraction): This transformation takes an expression in the individual and puts that expression in a function in a let…in block and replaces the expression with a call to that function.
4. CASE-DIST (Case distribution): This transformation takes a case expression inside a function call and moves the function call into each of the case code blocks. It can also change the scope of function definitions.
5. EMB (Embedding): This transformation changes the argument type of functions.
6. Crossover: This operator views a number of REQ transformations as alleles and recombines them using a genetic algorithm operating on fixed-length genomes.

When an individual is selected for expansion, a compound transformation is performed on it. A program that results from this compound transformation is evaluated for insertion into the kingdom. A compound transformation is a sequence of atomic transformations performed on the individual.

The number of compound transformations that are to be applied to a program is decided by a total work goal, called the cost limit, which is simply the number of individuals to be generated.

2.4 Summary of the ADATE search process

Note that ADATE runs indefinitely unless the user terminates its process. Figure 2 shows the top-level loop in ADATE, which also can be described as the following numbered step-by-step procedure.

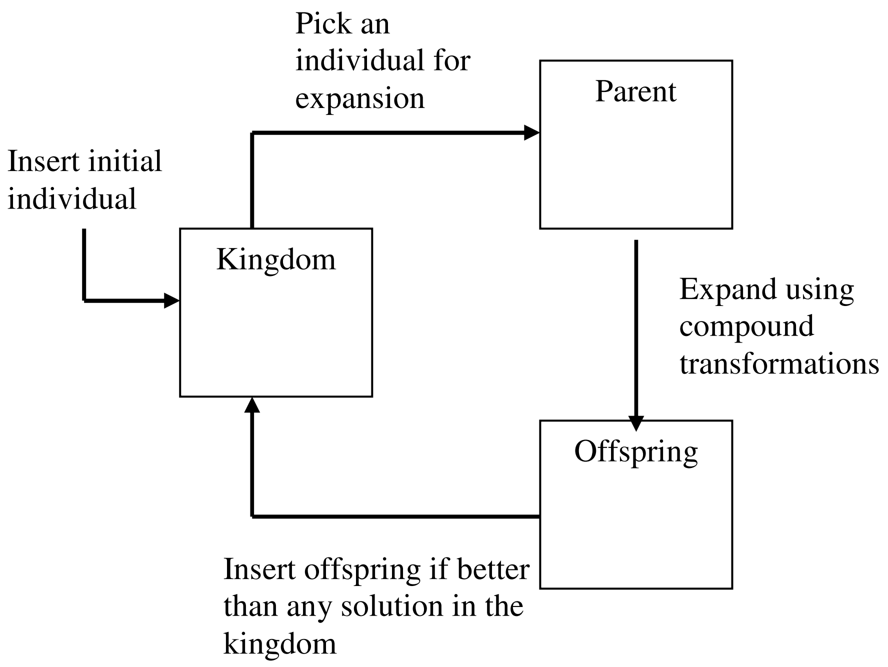

Pick an individual for expansion
Parent
Insert initial individual
Kingdom
Expand using compound transformations
Offspring
Insert offspring if better than any solution in the kingdom

Fig 2: Figure of the ADATE search process

1. Insert the initial individual into the kingdom
2. Pick an individual from the kingdom for expansion and choose a cost limit.
3. Generate a new individual with a compound transformation
4. Insert the new individual in the kingdom if it is to form a new family or fits into an existing family.
5. Repeat steps 3 and 4 a number of times based on the cost limit chosen for the expansion
6. Repeat steps 2 through 5 indefinitely

3. ADATE-ML

ADATE-ML is a subset of Standard ML that is used by ADATE's inferred programs. It is necessary to treat ADATE-ML in some depth as it is used when implementing problems.

It is an advantage if the reader is somewhat familiar with the Standard ML language. If this is not the case, an introduction and tutorial can be found at the Standard ML of New Jersey home page (http://www.smlnj.org//index.html).

ADATE-ML will be described by listing the differences from Standard ML. A grammar written in EBNF (Extended Backus-Naur Form) is also included for convenience and reference.

3.1 Differences from Standard ML

The main motivation for using a custom language instead of all of ML is to get a simpler language and one that is purely functional. The main features of ML that are left out of ADATE-ML are higher order functions, fully polymorphic typing, syntactic sugar (extra possibilities to write semantically equal code in different ways) and the module system.

This section will describe the differences between ADATE-ML and Standard ML by listing the ML constructs that are not supported. Examples of how to reformulate ML constructs to valid ADATE-ML code are given at the end of this section. Some constructs that are not supported in synthesized programs are:

- If … then … else expressions
- Boolean operators
- Curried functions
- Anonymous functions, i.e. lambda functions are not used
- Higher order functions
- Variable definitions using the keyword "val"
- Discriminators and selectors
- The _ token as a wildcard
- The patterns in case expressions are restricted
- User-defined polymorphic datatypes

There are some restrictions on function definition. ADATE-ML functions cannot be curried. Each function argument pattern is an n-tuple pattern with n>0. Each part of the pattern may be named using *as* expressions. The type of each function parameter must be given after the list of parameter names with type names separated by * tokens.

3.1.1 Reformulating ML to ADATE-ML code

This section gives a table with suggestions for how to implement the functionality of the ML constructs that are not supported in ADATE-ML.

ML Construct	ML Code Example	ADATE-ML Reformulation
if … then … else	if E then RHS1 else RHS2	case E of true => RHS1 | false => RHS2
Selectors	tl Xs	Case Xs of nil => raise NA1 | cons X'::Xs' => Xs'
Boolean operator	E1 andalso E2	case E1 of true => E2 | false => false

Left hand side	fun l nil = 0
	   | l(X1::Xs1) = 1+ l Xs1
		fun l Xs =
		  case Xs of
		    nil => 0
		  | cons(X1, Xs1) => 1 + l Xs1

Variable declaration in a let-expression	let val V = E1 in E2 end	case E1 of V => E2

Anonymous function	fun f Y = (fn X => X+1) Y	fun f Y =
		let
		  fun f1 X = X+1
		in
		  f1 Y
		end

The _ wildcard token	case x of 5 => 2 | _ => 3	case x = 5 of
		  true => 2
		| false => 3

Omitted as clause	datatype coord = point of int * int fun f (point(x,y)) = x	datatype coord = point of int * int fun f (C as point(x,y)) = x

3.2 Grammar

The grammar of ADATE-ML is written in EBNF (Extended Backus-Naur Form). EBNF is a notation for context-free grammars with terminals, non-terminals and productions. Note that some parentheses shown as mandatory in the grammar can be omitted.The parentheses are shown in this way to make the grammar shorter and easier to read. Also it is usual to have identifiers (names) and numbers as different non-terminals. In this grammar the <id> non-terminal covers them both.

```
<id> ::= ([a-Z]|[0-9]) { ([a-Z]|[0-9]) }

<type-cons> ::= <id> | <id> of <id> { * <id> }

<type-decl> ::= datatype <id> = <type-cons> { '|' <type-cons> }
```

```
<expr> ::= <id>
        | [<id>] '(' <expr> { , <expr> }  ')'
        | case <expr> of <rule> { '|' <rule> }
        | let <function-decl> in <expr> end
        | <expr> <id> <expr>

<pattern> ::= '(' <pattern> ')'
          | [<id>]  { , <pattern> }

<rule> ::= <pattern> => <expr>

<funpat> ::= '(' <funpat> ')'
          | <id> [ as <id> '(' <funpat> ')' ] { , <funpat> }

<function> ::=  <id> '(' [ <funpat> : <id> { * <id> }] ')' : <id> = <expr>

<function-decl> ::= fun <function> { and <function> }

<program> ::= { <type-decl> | <function-decl> }
```

3.3 Built-in types and functions

As mentioned there are datatypes and functions that are built into ADATE. "Built-in" in this context means that the datatypes and functions need not be declared in the ADATE-ML section of the specification. The built-in types are listed below as their definitions in the ADATE source code. They should mostly be self explanatory.

Note that infix operators cannot be declared in ADATE-ML code and are only used for some built-in functions. Please see the Standard ML Basis Library for documentation of the functions below.

```
type bool
val false : bool
val true : bool

type int
val = : int * int -> bool
val < : int * int -> bool
val ~ : int -> int
val + : int * int -> int
val - : int * int -> int
val * : int * int -> int
val quot : int * int -> int
val rem : int * int -> int

type real
val realEqual : real * real -> bool
val realLess : real * real -> bool
val realAdd : real * real -> real
val realSubtract : real * real -> real
val realUnaryMinus : real -> real
val realMultiply : real * real -> real
val realDivide : real * real -> real

val sigmoid : real -> real
val realFloor : real -> real
val realCeil : real -> real
val realTrunc : real -> real
val realRound : real -> real
val trunc : real -> int
val fromInt : int -> real
val sqrt : real -> real
val sin : real -> real
val cos : real -> real
val tan : real -> real
val asin : real -> real
val acos : real -> real
val atan : real -> real
val atan2 : real * real -> real
val exp : real -> real
val pow : real * real -> real
val ln : real -> real
val log10 : real -> real
val sinh : real -> real
val cosh : real -> real
val tanh : real -> real

val TCOUNT : int (* Current execution time. *)

(* The following type is an array that is not checked, that is no
exception is raised if it is read or written outside its dimensions
*)
type 'a uncheckedArray
val uncheckedArray : int * 'a -> 'a uncheckedArray
val uncheckedArraySub : 'a uncheckedArray * int -> 'a
val uncheckedArrayUpdate : 'a uncheckedArray * int * 'a -> 'a
uncheckedArray
```

4. Solving algorithmic problems with ADATE

This chapter explains how to implement problems with ADATE and give ADATE problem-specific information. Examples of such information are the training and validation sets. This chapter will describe in detail how to construct an ADATE specification file, which contains the information needed to automatically generate programs for an algorithmic problem.

As a user manual, the main focus of this text is the technical procedure of implementing problems. Important considerations to make when writing specifications are still given some space as this information is considered valuable for users of ADATE and is not fully documented elsewhere.

4.1 Specification

This section provides an overview of the contents and structure of an ADATE specification. After or during the reading of this section it may be helpful to examine the example specifications in Appendix B.

A specification describes the requirements for the function to be synthesized to the system. In other words, it tells ADATE what synthesized programs should do (not necessarily how they should do it). It consists of a text file containingADATE-ML as well as ordinary ML code. ADATE-ML is a valid subset of ML in which synthesized programs are written and is described in the previous chapter. The specification contains all the information that is specific to the problem being implemented. The code in the specification is embedded in ADATE's source code which is compiled with the MLton compiler.

The specification consists of two parts separated by the token '%%'. The section of the file before the '%%' token contains ADATE-ML code and the part after contains regular ML code. The reason for this division is that ADATE needs both code that can be used by the inferred function and code that it will use itself during the inference. The next two sub-sections will treat each of these two parts.

```
ADATE-ML code

%%

ML code
```

4.1.1 Contents of the ADATE-ML part of the specification

The ADATE-ML part consists of three elements. First, it contains data types and code that can be used by the inferred function. This code can be any valid ADATE-ML constructs. Secondly, the definition of the function to be induced must be given. This function must be named f. The f function usually just raises an exception, but f can be defined as any valid ADATE-ML function. The exceptions that f can use are predefined by ADATE and are named D0, D1, …, D9. Finally, this section must contain the function main. Even though it is f that is inferred it is main that takes the input for the problem that is implemented. The code for main should call f at least once and return the output for the problem. The main function is called by ADATE during evaluation value calculations. It may seem odd that f is not called directly, but sometimes it is advantageous to have main. For instance, the function f can be called several times and f may represent a small code block to be generated in a big main program. It is important to notice that it is only f that is the target of the induction.

4.1.2 Contents of the ML part of the specification

The second part of the specification (the part after the '%%' token) can contain any ML code the writer of the specification wishes. However, ADATE requires some particular functions and structures to be implemented. A specification, as mentioned previously, contains information that is specific to the problem being implemented. The way this works is that ADATE uses functions, lists and other values that are not implemented in the ADATE source code itself. This is a common approach for machine learning systems and such unimplemented functionality is often called "user call-backs". Note that the ADATE-ML code, which is a subset of ML, is also embedded in ADATEs ML code, so that the datatypes and functions are available also in the ML section of the specification.

The user callbacks of ADATE are described in detail in section 4.2. Both callbacks and the functions in the ADATE-ML section can also be seen as parameters to the system.

4.2 User callbacks

In this section, the user callbacks of ADATE are described in detail. As mentioned above, a user callback is an ML value (function, list or other value) that ADATE uses in its source code but leaves for the user to implement. The callbacks must be implemented in the ML part of the specification. The callbacks and a short description for each one is given before they are described in detail. Some of these callbacks are concerned with complex parts of the ADATE system that are not treated in this manual. Suggested default values for these callbacks are given. Each callback is named the way the value would appear in a specification.

1. **Inputs**: An ML list containing the training inputs
2. **Validation_inputs**: An ML list containing the validation inputs (if used)
3. **Abstract_types**: A list of types whose values can be inspected but not constructed.
4. **Funs_to_use**: The function set
5. **Reject_funs**: A list of functions that check for redundant code
6. **restore_transform**: A function that can alter individuals to accommodate a new f function
7. **Grade**: An ML structure used to grade the individuals for evaluation value calculations
8. **Output_eval_fun**: Function to evaluate individuals.
9. **Max_output_genus_card**: Number of individuals that can exist in an output genus.
10. **Max_time_limit and Time_limit_base**: Time limit on the running of individuals

4.2.1 Inputs and Validation_inputs

These values are ML lists containing the training and validation inputs. The training and validation inputs are given to ADATE as values of the datatype used as input to the function main. These are used by ADATE during training. It is not required that a validation set is used. If it is not, Validation_inputs should be an empty list.

4.2.2 Abstract_types

Abstract_types is an ML string list with the names of types whose values cannot be constructed. Abstract types can be useful under certain circumstances. For instance, a function should possibly not be allowed to alter a value that only serves as an identifier for elements in a list.

4.2.3 Funs_to_use

The specification needs to provide a function set containing functions that ADATE should try to use in synthesized code.. The set should contain auxiliary functions that may be needed including the constructors of the datatypes that are to be used. The function set is given as a list of strings.

4.2.4 Reject_funs

Reject_funs is a list of functions that is used by ADATE during synthesis of expressions. There are many expressions that are semantically equivalent. Consider for instance, the expressions 1*1, 2*0 and nil @ [1,2,3]. All these expressions are valid in ML, but should not be allowed. In addition to bad appearance such constructs will increase the workload of ADATE and lead to less efficient inferences. When ADATE synthesizes an expression, it calls each function in the Reject_funs list and rejects the expression if any of them returns true.

4.2.5 restore_transform

ADATE contains functionality to resume a run from a checkpoint file even if one has changed the definition of f. For this to be possible ADATE needs some means to change old individuals so that they comply with the new definition of f. The function restore_transform may be implemented by the user to do this.

4.2.6 Grade

ADATE requires the user to provide the means of grading the individuals during evaluation value calculations. The callback is defined as an ML structure and consists of all information that is needed to calculate and evaluate the grade of an individual. This is but one of several means ADATE uses for calculating an individual's evaluation value and it is usual to let the user-defined grade return the same value for all grade calculations, that is, not use this feature of ADATE.

The structure contains the following values:

1. **grade**: The type of grades
2. **zero**: A grade value representing zero.
3. **op+**: An operator that adds two grades.

4. **comparisons**: A list of functions that compare two grades. A function returns a value of the ML type order. This type can take three values, namely LESS, GREATER or EQUAL.
5. **toString**: A function that gives a string representation of a grade
6. **pack**: During induction, ADATE needs to represent grades as strings to send them across networks or store them in checkpointing files. This function takes a grade and returns such a string.
7. **unpack**: A function that takes a string made with the pack function and calculates the corresponding grade.
8. **post_process**: A function that can change a grade from one type to another
9. **toRealOpt**: Should always be left at the default value NONE.

The grade type may be any ML type, but it is common to use int or real. ADATE treats the smallest (as returned by the comparisons) of two values as the best grade.

It is perhaps easiest to see how this works by considering a grade using the int type. Here the value zero is the number 0 and the + operator is arithmetic addition. The grading scale may then be from 0 (a perfect grade) and upwards. The comparison function will be the familiar comparison function for integers found in most programming languages.

The pack and unpack functions are used by ADATE when writing checkpoint files (see 5.3.1). One can think of this as a kind of serialization. Notice that the toString function is intended to provide a string representation that looks good when printed.

It is also possible to change the grade measure when one wishes to resume an ADATE run from checkpoint files. This works in about the same manner as the changing of the function definition as described above. The function to transform the old grades to new grades is the post_process value.

The toRealOpt value is reserved by ADATE for future use and must be left at NONE.

4.2.7 Output_eval_fun

The output evaluation function is given the input data and output data for one particular individual on one particular training or validation example. In addition it gets the index in the training set of the example in question. The definition of the function for the XOR problem is given in Appendix B. One may notice that the validation examples are appended to the Inputs list and that the index always refers to the position of the example in this concatenation of training and validation inputs.

The output of the function is used to calculate the evaluation values of the individual on the problem instance. The output consists of a record with two integers, which are always positive, representing the correctness or incorrectness of the individual and a grade as defined by the grade structure.

4.2.8 Max_output_genus_card

Max_output_genus_card says how many individuals that should exist in an output genus for a given base individual (see, 2.2.2). It is usual to only allow a few individuals. (If more were present some promising code blocks may not be lost, but it would also require more work to maintain many.) This is an heuristic choice and its effects are not well understood or known.

4.2.9 Max_time_limit and Time_limit_base

Max_time_limit and Time_limit_base are measures on the time an individual is allowed to execute before it is terminated during evaluation value calculations. ADATE measures time as the number of function calls and case-analyses made by the individual (as opposed to regular time). Max_time_limit is a number which is the maximum time complexity an individual may use for any of the examples it is run with. This is needed mostly to avoid infinite recursions and terminate programs that require more time than what is assumed to be necessary for any of the training examples for any reasonable program.

ADATE can use many time limits simultaneously. The time limits that will be used in the size-time grid are Max_time_limit / Time_limit_base ^ i for all i = 0, 1, 2, … that give a time limit greater than or equal to one. For instance, if the base is 2 and the maximum limit is 64 then the time limits 1, 2, 4, 8, 16, 32 and 64 will be used.

4.3 Guidelines for writing specifications for ADATE

As the specification encircles the portion of the solution space that is accessible to ADATE during induction, care needs to be taken in writing specifications. This section will try to outline some important considerations to keep in mind while implementing problems with ADATE.

As ADATE is a relatively young system, the effects of combinations of parameters in the specification are not yet thoroughly researched and understood. Knowledge about parameters for ADATE (and most EC in general as well) is based on experience. This section should therefore be seen as practical advice about problem implementation based on experiences made by Roland Olsson and his students during use of the ADATE system and not as scientifically documented facts.

Readers may also find “The art of writing specifications for the ADATE automatic programming system” by Roland Olsson useful. There are also many general properties of specifications for machine learning that are applicable to ADATE.

4.3.1 General advice

An overall principle to follow while writing specifications is to try to keep things simple. Superfluous information tends to increase the workload of the inference. It is on the other hand important that the information is adequate to describe the problem or it might be impossible, or at least more difficult, to find valid and reliable solutions. It is in practice often non-trivial to write specifications that are simple, unambiguous, concise and precise. This is perhaps especially so when the problem is not well understood or a solution is not known. Therefore, specifications are often written with some unnecessary information to ensure possible and correct inferences.

Also it can be advantageous to remember that ADATE performs a learning process. One might view ADATE as a child that knows nothing about its environment and will experience it through training. A specification will profit from being pedagogical. Pedagogy is the study of teaching methods and theory about how learning takes place. It is relevant to ADATE as its learning can be made more efficient by following pedagogical principles. One should keep in mind how ADATE learns and write the specification to make gradual and evolutionary learning possible.

4.3.2 The training set

The training set is perhaps the most important part of the specification. It is decisive for the generalizing ability as well as describing the problem to ADATE. A training set contains inputs of problem instances. Foremost, the set should cover sufficiently many cases of the inputs. A case in this context is a set of inputs that would be treated the same by a correct program. For instance, consider the problem of moving one box inside another and the question in what direction the box would have to move. A case here would constitute all inputs where the box that is to be moved is above the other box. In this case a solution would have to move the box down. Also, a general principle is to include the simplest inputs of the problem and overall have a larger number of simple problem instances than difficult ones. In fact, it is advantageous to have few large size inputs. Some big inputs are still recommended to avoid programs with large time complexity, overfitting and rote learning.

The lack of coverage of all input cases may lead to overfitting. This is because inferred programs are likely to give incorrect answers when encountering cases not present in the training set. Simple inputs are important because ADATE is likely to find programs for these faster than for large inputs and thereby speed and direct the start of the evolution. This is also why it is important to have many simple inputs. ADATE identifies programs that classify these instances correctly as having a high evaluation value. Lack of simple instances may very well lead to early convergence or inefficient inference. The size of the inputs as well as the size of the training set is naturally of importance for the efficiency of the induction as it typically takes longer time to compute evaluation values for large inputs and training sets. Ideally, the training set should be just large enough to cover all cases and have a progression of input size that facilitates efficient learning. This is of course seldom achieved in practice and it is, as mentioned, common to introduce more examples than believed necessary to get reliable results.

4.3.3 The function set

The function set contains, as mentioned above, functions and constructors that ADATE can use in inferred programs. The set is therefore an efficient way to set bounds on the search space. ADATE may invent functions, but not datatypes, during induction, but an adequate function set might lead to more efficient inference. If the function set, for instance, contains the exact help functions needed to solve a problem the inference of a correct solution should be faster. Also, depending on the datatypes, an inference might be infeasible with an insufficient function set. In the case of the example on moving one box inside another, assuming that the boxes are represented as coordinates in a Cartesian coordinate system, a solution cannot be found unless the function set provided the '<' function or some other means to compare the coordinates of the boxes. If not, ADATE could not find the positions of the boxes relative to each other. The function set could both increase and decrease the workload of the inference. A large function set increases the search space. But depending on ADATE to induce help functions may lead to big programs that again might take more time to find. Also, by using a small function set, one limits the possible solutions ADATE may find.

4.3.4 ADATE-ML code issues

There are some things to think about regarding the ADATE-ML code in the specification. The type specifications should be precise and constitute an unambiguous grammar. It is an advantage if the grammar gives small sentences and that the number of valid sentences is small and final. This should keep the number of valid programs using the datatypes small and thereby reduce the search space. This is naturally not always possible. A common example is programs that require lists. Help functions written in the ADATE-ML section will frequently be used during evaluation value calculations.

4.3.5 The output evaluation function

ADATE uses the evaluation value to decide which individuals should be kept in the kingdom and allowed to breed. One must therefore make sure that ADATE has sufficient grounds for this. This means that the evaluation value measure is fine-grained enough to notice program improvements. However, note that a too fine grain might be misleading and lead to a less efficient inference. On the other hand if it is coarse, some programs that would have lead to a faster solution may never be found.

4.3.6 Representing the problem to ADATE

By representation of a problem we think of how it is described to ADATE. For example, if we have a three-dimensional box that is part of a problem it can be represented by the 8 coordinates of its corners or only by the bottom-left and top-right corners. In fact, all datatypes defined in the ADATE-ML section gives a representation of something to ADATE. Representation is especially important in EC as a compact representation leads to a small search space. In the case of the three-dimensional box it is clearly beneficial to use the representation with only two coordinates.

4.4 Example specifications

This section briefly describes the example specifications in Appendix B. It is intended that the specifications should show basic uses of the ADATE system. A bullet list of things to specially take note of in each specification is given in the following descriptions.

The specification shows methods that are appropriate for many problems and they might be used as recipes for problem implementation. They are also examples of ADATE-ML programming.

4.4.1 The XOR specification

This specification was written purely for pedagogical purposes. Note that both the datatype and help function could have been omitted. If these two values are removed the specification can be used as a skeleton for input-output pair specifications. One only need to redefine f and main, change the training -and output set and update the function set accordingly. The main purpose of the XOR specification is to show:

- An example of an input-output pair specification
- The grade structure when not using a grading system
- An output evaluation function for input-output pairs
- Values for user callbacks that can be regarded as default values for specifications.

4.4.2 The Boxbox specification

This is a specification for a problem in which a box in three-dimensional space is to be moved inside a container (also a box).[1] The problem is described using analytic geometry (the familiar geometry learned in elementary school). The moving box can only move along the axes of the coordinate system (6 directions) and the movement is performed in steps of length one.

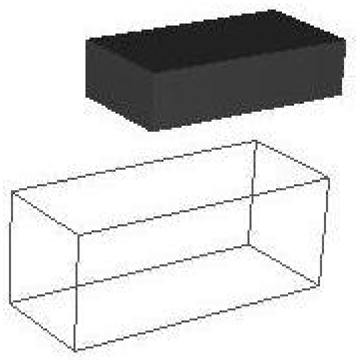


Fig 3: An instance of the boxbox problem

This problem shows a more complex specification that makes use of a grading system. It also uses more datatypes and help functions. When examining the specification one may take note of:

[1] This specification is not the same as the specification used for the boxbox problem in the thesis referenced in Appendix A. Note that grading in the case of the boxbox problem is not necessary. ADATE has no problem finding a solution without it.

- Use of main
- Use of an integer grading system
- Output evaluation function for grading with integers and not using input-output pairs

5 Running ADATE

This chapter describes the procedure of compiling and running problems with ADATE. It also explains the prerequisites (setup of the system) before problems can be compiled. Before an ADATE run can be carried out a specification for the problem to be solved must have been made. How to make a specification is described previously in this manual.

We also try to give a troubleshooting guide for some of the most common errors made by new users.

The example specification for the induction of the XOR function is used in this chapter for demonstration of shell commands. The procedure is always the same regardless of the specific specification, so the only change that needs to be made for other specifications is the name of the specification file.

5.1 Set-up of the system

ADATE needs to run on a Linux system using x86 CPUs.[2] It is important to notice that ADATE cannot run on systems with other CPUs. The reason for this is that ADATE represents its individuals as x86 machine code during evaluation value calculations.

The current distribution of ADATE must be installed on the system. This distribution can be obtained from the ADATE homepage at HiØ and consists of a compressed tar archive (.tgz format). To install ADATE one simply needs to extract the archive. The distribution should contain the following files that must be placed in the same folder: makespec, main1.sml, main2.sml and mlt. It also contains both an RPM distribution and a .tgz distribution of a suitable version of the MLton compiler (http://www.mlton.org). If you use the .tgz distribution, you need to set the lib variable in the mlton script to point to the installation directory. MLton requires the GNU multiprecision library, for which RPM files also are included in the distribution.

ADATE may also be run on a Linux cluster under MPI. ADATE has been thoroughly tested using the MPICH implementation of MPI and it is therefore recommended that it is used for running ADATE on clusters. We do not teach running ADATE on many processors in this manual even if that is desirable to run all but the smallest problems. If you wish to do this and have trouble making it work properly you are welcome to contact Roland Olsson.

[2] A Microsoft Windows version for an earlier version of ADATE is available, but it is somewhat unstable and we do recommend the use of the Linux version. The procedure to run ADATE is different for the Windows version and is not treated in this manual.

5.2 Compiling ADATE

ADATE consists of ML code and is compiled for each individual problem using the mlton compiler. This section describes how to get an executable from a specification file. First, a quick step-by-step procedure is given and then an example compilation using the XOR specification.

5.2.1 Step-by-step procedure for compiling ADATE

As mentioned previously, the code in the specification is to be embedded in the ADATE source before it is compiled. However, the makespec program must process the specification before it can be included in the ADATE source. The result from makespec should be an sml file that is named the same as the specification file with the suffix '.sml'. This file is then included in ADATE by inserting it between the main1.sml and main2.sml files. Then ADATE can be compiled with mlton.

The next section shows an example compilation with shell transcripts using the XOR example specification.

5.2.2 Example compilation

Assuming that the specification file XOR.spec and the ADATE files makespec, main1.sml, main2.sml and the script mlt is located in the folder ADATE, the following shell transcripts compile the XOR problem from the XOR.spec specification. The transcripts are marked with grey boxes.

Run the makespec program on the specification file:

```
[geirvatt@localhost ADATE]$ ./makespec XOR.spec


Processing XOR.spec
[geirvatt@localhost ADATE]$
```

This should leave a file named XOR.spec.sml in the ADATE folder. This file is then included in the ADATE source between main1.sml and main2.sml. This can be done with the cat command like this:

```
[geirvatt@localhost ADATE]$ cat main1.sml XOR.spec.sml main2.sml >
main.sml
[geirvatt@localhost ADATE]$
```

This puts the final source code for the XOR problem in the main.sml file. Note that the file must be named main.sml. This file can then be compiled with mlton. This is done with the script mlt that simply runs mlton with the necessary parameters:

```
[geirvatt@localhost ADATE]$ ./mlt
```

This leaves the executable main in the ADATE folder and is the finished ADATE executable for the XOR problem. See the next section for how to run it. The mlton compiler will give a lot of warnings when compiling. This is normal and does not present a problem.

5.3 Running ADATE

To run ADATE, the compiled executable main must be invoked. The executable takes one command line argument. When starting an inference this parameter must be start. A shell transcript is given:

```
[geirvatt@localhost ADATE]$ ./main start
```

ADATE will print out some information before the induction starts. If the start was successful, the ADATE log, trace and validation files for the run should have been created in the working directory. Chapter 6 describes how to interpret them.

ADATE will now run indefinitely until the user terminates the run (by killing the process). By monitoring the output files, the user can oversee the evolution to determine when to end the run. Use the UNIX command nohup to ensure that the ADATE process is not killed automatically by the operating system as soon as your login session ends, for example if an SSH connection is broken.

Also, check that the stack size is unlimited using the command line “ulimit –s”. Since a lot of log data is printed to the standard output, it may be best to redirect the output of an ADATE run to a file using “>& log” to prevent the terminal window from hanging due to too much data being printed.

5.3.1 Resuming inferences

During a run, ADATE will produce checkpoint files. A checkpoint file is a file in which ADATE stores the state of the induction at the time the file is made. If the ADATE run is terminated for some reason, the run may be resumed from a checkpoint file.

The checkpoint files are named after the prefix of the specification file, the number of the checkpoint and the process id of the ADATE run. An example in the case of the XOR specification could be:

```
XOR.0.24567
```

When ADATE resumes inferences, it assumes that the checkpoint file is named after the prefix of the specification file followed by '.0.pop'. So when ADATE is to resume from a specific file this should be renamed accordingly. In the case of the XOR checkpoint file mentioned above this would be:

```
[geirvatt@localhost ADATE]$ mv XOR.0.24567 XOR.0.pop
```

Then ADATE can be resumed with:

```
[geirvatt@localhost ADATE]$ ./main restart
```

5.4 Troubleshooting

New users, as well as experienced ones, may encounter problems when implementing and running problems with ADATE. In this section, we try to list some of the most common errors and problems, and their typical resulting symptoms..

The focus is mainly on runtime problems and problems that cause unwanted properties of the evolution. The mlton compiler has good error checking to catch compilation and linking errors. Note, however, that the makespec ADATE-ML parser does not catch all syntax errors in the ADATE-ML part of the specification and does not do semantic error checking (such as variable declarations before use etc.). If the erroneous ADATE-ML code is still valid ML code the errors will also not be captured by mlton. This is the number one reason for misleading runtime exceptions.

The section on common problems may also be used as a checklist before compiling. The mlton compiler is somewhat slow and a compilation takes about five minutes on a 1.7 GHz Pentium IV.

5.4.1 Common problems

Problem	Usual symptoms
Constructors for datatypes in the ADATE-ML section are not present in the function set.	As synthesized programs cannot use them, the problem may be unsolvable.
Corrupted main files. This can happen if the cat command is used incorrectly (by for instance using the spec instead of spec.sml file.	Misleading mlton compilation errors.
Insufficient grounds for ADATE to	The evolution will converge prematurely.

notice improvements in evaluation value. This can be due to either a too small training set or a too coarse grading scale.	
ADATE was invoked without the start parameter	Exception Match
A function or constructor in the function set does not exist	Exception Unknown_fun_to_use
Invalid function definition (omitted as clause)	Exception Vars_in_pure_tuple_pat_exn
A constructor is misspelled later in the code.	Exception Constructor_table_exn.

6 Reading the ADATE output files

ADATE reports data about the induction in a series of files that will be located in the directory ADATE was started in. This includes information about individuals, the kingdom and the evolution in general. All the data in the files will not be explained. This is because they contain some debug information for the system and that some data is concerned with aspects of the system that are not treated in this manual.

The three main files that are described in this chapter are the log, trace and validation files. They can be identified by their name, which consists of the prefix of the specification file and the type of the file: log, trace or validation. The suffix of the files is the process id number (pid) for the run. For instance, a run with the XOR specification with process id 18596 will generate the following files:

```
XOR.log.18596
XOR.trace.18596
XOR.validation.18596
```

- The log file prints out information about the individuals
- The trace file prints out data about the kingdom.
- The validation file contains print-outs to check for overfitting.

The reader will be able to better utilize this chapter if he understands the basic workings of ADATE as described previously in this manual. Especially the kingdom management and transformations (genetic operators) are important. Transcripts from the files are given but it might be advantageous to examine complete files while reading this chapter.

6.1 Basic information about individuals that appear in all output files

Throughout the output files, an individual is presented as a single line of numbers. These numbers give information about how the individual performed either over the training or over the validation set, the time it took to run it on the set, syntactic complexity of the individual and means for uniquely identifying it. Note that some information is the same regardless of which set is used to calculate the data. An example transcript from a log file is given below with an explanation of each number in the line.

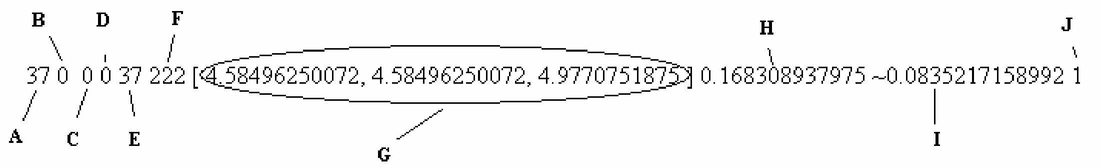

```
37 0  0 0 37 222 [4.58496250072, 4.58496250072, 4.97707518755] 0.168308937975 ~0.0835217158992 1
```

Fig. 4: This transcript is taken from a log file and shows information about how the individual performed on the training set.

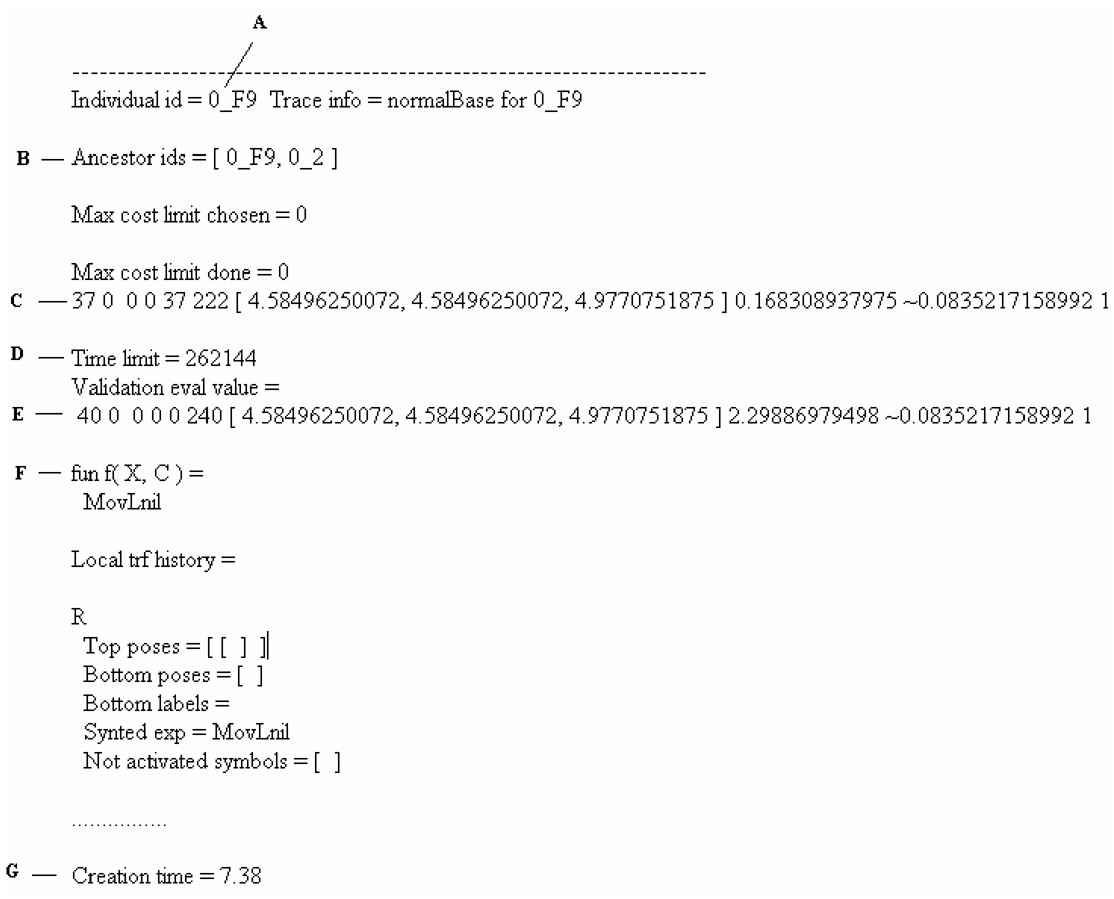
A. The number of examples that were correctly solved by the individual.
B. The number of examples for which either too much memory or too much time was used.
C. The number of examples that the individual didn't solve correctly.
D. The user-defined grade. This is the return value of the Grade.toString function which typically is an empty string if no grade is no used. In that case, this field will not exist.
E. The number of examples that are truly correct. Without further explanation, this indicates that ADATE is more confident about the correctness of these examples.
F. The time the individual used to execute on all examples (As defined by ADATE's built in time complexity measure)
G. These numbers are syntactic complexity measures (sizes of the individual in bits).
H. A semantic fingerprint that identifies individuals which are semantically equal to this one.
I. A syntactic fingerprint that uniquely identifies the individual.
J. The number of super-combinator functions in the individual, also counting the induced function f.

This line will simply be referred to as 'basic information' in the rest of the manual.

6.2 The log file

The log file prints information about individuals that are generated during the ADATE run and inserted into the kingdom. Basic information about all individuals inserted into the kingdom is given.

Before such an individual is printed in its entirety, ADATE inserts a line of only the '-' character as a separator. We give a transcript of an individual from a log file below. The data about the individual are then described. Note that not all data is treated. This is because it deals with things not covered in the manual.



```
                    A
                   /
------------------/------------------------------------------------
Individual id = 0_F9  Trace info = normalBase for 0_F9

B — Ancestor ids = [ 0_F9, 0_2 ]

    Max cost limit chosen = 0

    Max cost limit done = 0
C — 37 0  0 0 37 222 [ 4.58496250072, 4.58496250072, 4.9770751875 ] 0.168308937975 ~0.0835217158992 1

D — Time limit = 262144
    Validation eval value =
E —  40 0  0 0 0 240 [ 4.58496250072, 4.58496250072, 4.9770751875 ] 2.29886979498 ~0.0835217158992 1

F — fun f( X, C ) =
      MovLnil

    Local trf history =

    R
      Top poses = [ [  ]  ]
      Bottom poses = [  ]
      Bottom labels =
      Synted exp = MovLnil
      Not activated symbols = [  ]

    ................

G —  Creation time = 7.38
```

Fig 5: Excerpt from an ADATE log file showing information about an individual.

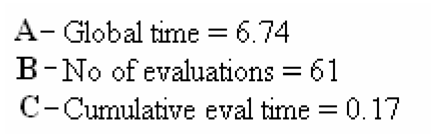
A. An unique identifier of the individual
B. A list of the ancestors of the individual (e.g. a trace of compound transformations back to the initial individual in the kingdom).
C. Basic information over the training set (see 6.1).
D. The time limit for the individual in the size-time grid
E. Basic information over the validation set (see 6.1).
F. The program as ADATE-ML code
G. The creation time of the individual in CPU time seconds. Note that this time may be misleading if ADATE is running on a cluster.

The Local trf history in the transcript gives information about the transformations that created the individual. However, it would be too extensive and complex to explain this here. The rest of the log before the next individual with detailed information is printed is not relevant for casual ADATE users.

6.3 The trace file

The trace file concerns itself with the kingdom and how it is organized in ADATE. First, it gives some statistics about the time it took to perform different tasks on the individuals in the kingdom. Then it shows the time-complexity grids and finally lists individuals that are queued for expansion (breeding).

The statistics is mostly useful for debug information and to more advanced users, but three time measures will be mentioned here. A transcript from a trace file follows.



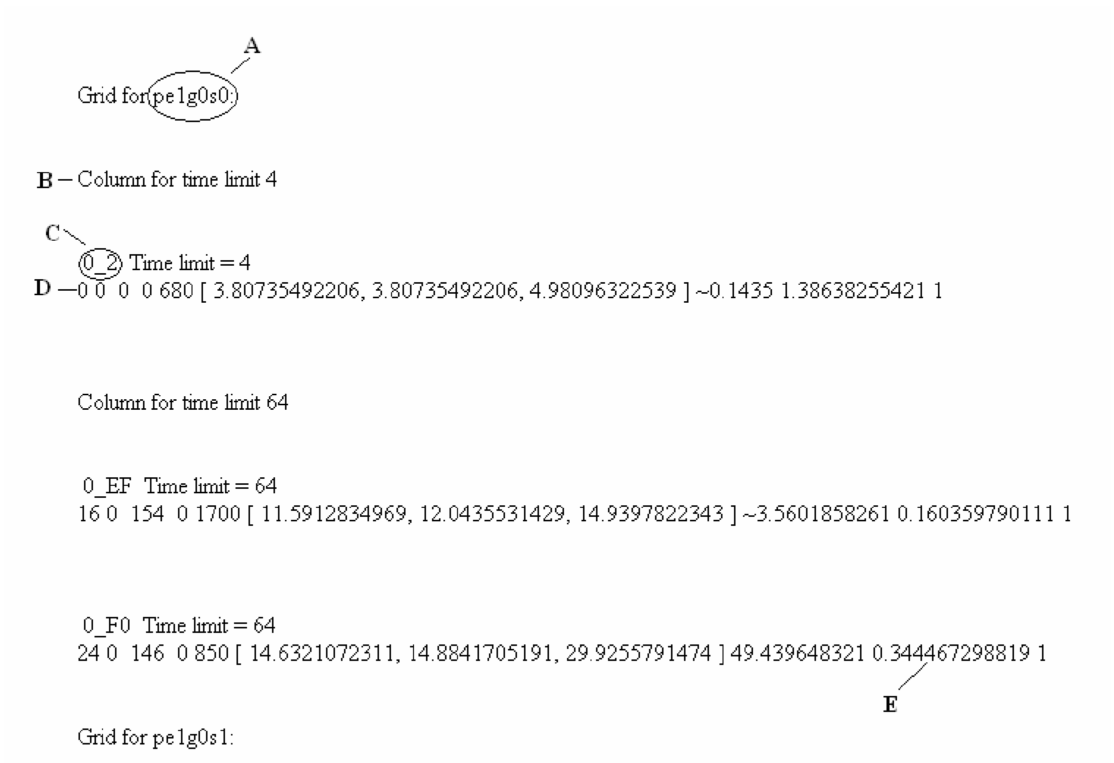
```
A- Global time = 6.74
B -No of evaluations = 61
C-Cumulative eval time = 0.17
```

Fig 6: A transcript from a trace file showing measures of creation time

A. The time it has taken to generate the kingdom since the start of the induction.
B. The total number of evaluations (evaluation value calculations) that has been performed.
C. The cumulative time used for evaluation value calculations.

The size-time-complexity grids are then listed. It is in the grids one can identify the individuals in the kingdom. They are in the trace file given as basic information lines (see 6.1). A transcript of a grid from a trace file is given below.

It is recommended to use the grid pe1g0s0 as the evaluation function pe1 gives highest priority to the number of correct answers and since an individual printed in one grid is not printed in the subsequent ones.



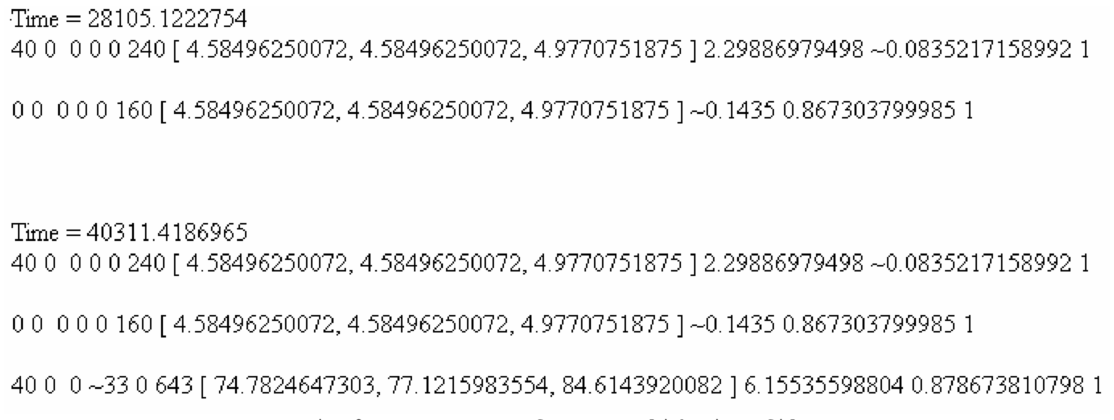
```
Grid for pe1g0s0:

B – Column for time limit 4

 0_2 Time limit = 4
D –0 0  0  0 680 [ 3.80735492206, 3.80735492206, 4.98096322539 ] ~0.1435 1.38638255421 1


Column for time limit 64


 0_EF  Time limit = 64
16 0  154  0 1700 [ 11.5912834969, 12.0435531429, 14.9397822343 ] ~3.5601858261 0.160359790111 1


 0_F0  Time limit = 64
24 0  146  0 850 [ 14.6321072311, 14.8841705191, 29.9255791474 ] 49.439648321 0.344467298819 1

Grid for pe1g0s1:
```

Fig 7: Transcript from trace file of a time-complexity grid.

A. This identifies which grid is printed. It consists of three parts: The first part tells which program evaluation function has been used and can be pe1, pe2 or pe3. The second part tells which grade measure has been used. If the specification contains only one grade comparison function, this is always g0. Finally, the third part tells which syntactic complexity measure has been used and is s0, s1 or s2.
B. Tells that it will list the individuals in a particular column in the size-time grid next in the trace file. The individuals are listed in order of increasing syntactic complexity.
C. This is a unique identifier for the individual that also is printed in the log file.
D. The basic information about the individual (see 6.1)
E. As the individuals are printed in the order of syntactic complexity in the grid, it is the individual printed last in each grid that is the individual with the highest training evaluation value (given the measures of the grid, see A). Its syntactic fingerprint can be used to find it in the log file.

6.4 The validation file

The validation file regularly lists all individuals for the highest time limit for the pe1g0s0 grid. The individuals are sorted after training evaluation value with the best individual on the training set given last, but only validation values are printed. Before each new listing, an overall time measure is printed. Note that this includes all CPUs if ADATE is running on a cluster. An example transcript from a validation file with two listings is given below.



```
Time = 28105.1222754
40 0  0 0 0 240 [ 4.58496250072, 4.58496250072, 4.9770751875 ] 2.29886979498 ~0.0835217158992 1

0 0  0 0 0 160 [ 4.58496250072, 4.58496250072, 4.9770751875 ] ~0.1435 0.867303799985 1



Time = 40311.4186965
40 0  0 0 0 240 [ 4.58496250072, 4.58496250072, 4.9770751875 ] 2.29886979498 ~0.0835217158992 1

0 0  0 0 0 160 [ 4.58496250072, 4.58496250072, 4.9770751875 ] ~0.1435 0.867303799985 1

40 0  0 ~33 0 643 [ 74.7824647303, 77.1215983554, 84.6143920082 ] 6.15535598804 0.878673810798 1
```

Fig 8: An excerpt from a validation file

The main use of the validation file is to check the generalizing ability of the results from the induction. As the individuals are sorted by their training evaluation value, the individuals should still appear sorted by validation value if there is no or little overfitting.

6.5 Finding information in the files

It may be tedious to learn and understand everything about the output files to find out how to locate specific information. Also, one might have to interpret different information in the files to find what one is looking for. This section is included for convenience and describes how to find some of the most important and most used information in the files. Section 6.1 about basic individual information must in any case have been read and understood.

6.5.1 The best individual

By the best individual one usually think of the best validated individual. This can be found by looking at the bottom of the validation file. Use its syntactic fingerprint to search for it in the log file.

If one would like to find the individual with the highest evaluation value this can be found in the trace file. Search for the string 'pe1g0s1' from the end of the file. The first individual listed (as a basic information line) above it is the individual with the best evaluation value over the entire run.

6.5.2 Training and validation data of an individual

These are both listed in the log file (see fig 5)

6.5.3 Generalization

It is easiest to check for generalization abilities in the validation file. The individuals should appear to be sorted by the number of correct classified examples and the grade.

It is also possible to check for overfitting in the trace file. It is a sign of overfitting when the differences in evaluation value between adjacent individuals are small.

6.5.4 Creation time of an individual

There are several measures of the creation time of an individual in the files. If ADATE is running on a single CPU, the best measure of creation time is in the log file, see the letter G in fig 5. If ADATE runs on a cluster this is not as useful, as it will only measure the CPU time on the node that created the program. The best estimate for overall computational work that was required to find an individual may then be obtained by searching for the individual in the validation file. The first time listed above it is proportional to the creation time on all CPUs together.

Table of figures

Appendix A: Resources

ADATE specific resources

The ADATE homepage at Østfold University College. Contains all articles about ADATE published by Roland Olsson and also other useful information.
(http://www.ia-stud.hiof.no/~rolando/)

ADATE – et system for automatisk programmering.
A presentation, written by Henrik Berg, of the ADATE system. Parts of chapter 2 is based and/or copied from it.
(www.ifi.uio.no/dbsem/foils2004vaar/adate.pdf)

Homepage for a MS thesis, by Geir Vattekar, that deals with the use of ADATE in three-dimensional environments. The thesis contains several specifications (from simple to complex). Parts of this manual are based and/or copied from the thesis:
(http://www.ia-stud.hiof.no/~geirvatt/)

Machine learning and EC resources

The Hitch-Hiker's Guide to Evolutionary Computation (
http://www.etsimo.uniovi.es/pub/EC/FAQ/www/top.htm)

Usenet newsgroups dealing with artificial intelligence (comp.ai.*)
(http://groups.google.no/groups/dir?sel=33583203&expand=1)

YAHOO ML Index
(http://dir.yahoo.com/Science/computer_science/artificial_intelligence/machine_learning/)

The Machine Learning network Online Information Service
(http://www.mlnet.org/)

Books:

Banzhaf, Wolfgang et al. (1998), Genetic Programming – An Introduction, Morgan Kaufmann Publishers, Inc.

Mitchell, Tom (1997), Machine Learning, McGraw-Hill international editions.

ML resources:

Standard ML of New Jersey
(http://www.smlnj.org/)

Online ML tutorial
(http://www.dcs.napier.ac.uk/course-notes/sml/)

The mlton compiler homepage
(http://mlton.org)

Appendix B: Example Specifications

XOR Specification:

```
fun not( a : bool) : bool = case a of false => true | true => false

fun f (  ( X, Y ) : bool * bool  ) : bool = raise D0
fun main (  ( X, Y ) : bool * bool ) : bool  =  f ( X, Y )

%%

val Inputs = [ (false, false),  (false, true),
               (true, false),  (true, true) ]
val Validation_inputs = [ ]

val Outputs = Vector.fromList( [ false, true, true, false ])

val Abstract_types = []

val Funs_to_use = [ "not", "false", "true" ]

val Initial_program = NONE

val Reject_funs = []
fun restore_transform D = D

structure Grade : GRADE =
struct

type grade = unit
val zero = ()
val op+ = fn(_,_) => ()
val comparisons = [ fn _ => EQUAL ]
val toString = fn _ => ""
val pack = fn _ => ""
val unpack = fn _ =>()

val post_process = fn _ => ()

val toRealOpt = NONE

end

fun output_eval_fun( I : int, _ : input, Y : bool )  =
   if  Vector.sub( Outputs, I ) <> Y  then
     { numCorrect = 0, numWrong = 1, grade = () }
   else
     { numCorrect = 1, numWrong = 0, grade = () }

val Max_output_genus_card = 2

val Max_time_limit = 65536
val Time_limit_base = 2.0
```

Boxbox Specification:

```
datatype point = point of int * int * int
datatype box = box of point * point
datatype world = world of box * box

datatype action = finished | up | down | left | right | foreward | backward

datatype mainReturn = mainReturn of world * int

fun moveBox( (act, Box as box( P1 as point(x1, y1, z1 ), P2 as point( x2, y2, z2) ) )
        : action * box ) : box =
                case act of
                  finished => Box
                | up => box( point(x1, y1+1, z1), point(x2, y2+1, z2) )
                | down => box( point(x1, y1-1, z1), point(x2, y2-1, z2) )
                | left => box( point(x1+1, y1, z1), point(x2+1, y2, z2) )
                | right => box( point(x1-1, y1, z1), point(x2-1, y2, z2) )
                | foreward => box( point(x1, y1, z1+1), point(x2, y2, z2+1) )
                | backward => box( point(x1, y1, z1-1), point(x2, y2, z2-1) )

fun nextWorld( (act, World as world( Box1, Box2 ))
        : action * world ) : world =
                world( Box1, moveBox( act, Box2 ) )

fun f(  (World as world(
                                        Box1 as box( cp1 as point(x1,y1,z1), cp2 as point(x2,y2,z2) ),
                                        Box2 as box( p1 as point(x1',y1',z1' ), p2 as point(x2',y2',z2') ) )
                    ): world ) : action = raise D0

fun main( Ret as mainReturn(World, I) : mainReturn ) : mainReturn =
        case f( World ) of
          finished => mainReturn(World, I)
        | up => main( mainReturn ( nextWorld( up, World ), I+1) )
        | down => main( mainReturn( nextWorld( down, World ), I+1) )
        | right => main( mainReturn( nextWorld( right, World ), I+1) )
        | left => main( mainReturn( nextWorld( left, World ), I+1) )
        | foreward => main( mainReturn( nextWorld( foreward, World ), I+1) )
        | backward => main( mainReturn( nextWorld( backward, World ), I+1 ) )

%%

val Inputs = List.map (fn X => mainReturn( X, 0)) [
        world( box(point(1,3,2),point(4,7,8)), box(point(1,3,2),point(2,5,5)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(2,4,5),point(3,6,8)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,4,3),point(2,6,6)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(3,5,3),point(4,7,6)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(2,4,4),point(3,6,7)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(3,5,2),point(4,7,5)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(4,3,3),point(5,5,6)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,6,2),point(2,8,5)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,3,6),point(2,5,9)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(0,3,2),point(1,5,5)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,2,3),point(2,4,6)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(2,3,1),point(3,5,4)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(0,2,3),point(1,4,6)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(2,1,~1),point(3,3,2)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(0,4,1),point(1,6,4)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(4,7,4),point(5,9,7)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(2,6,7),point(3,8,10)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(4,4,5),point(5,6,8)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(0,2,1),point(1,4,4)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(3,1,~1),point(4,3,2)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(0,1,0),point(1,3,3)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(4,6,6),point(5,8,9)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,7,8),point(3,9,11)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~1,4,3),point(0,6,6)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,0,2),point(2,2,5)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(3,8,5),point(4,10,8)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(1,4,~2),point(2,6,1)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(3,3,10),point(4,5,13)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~1,0,~3),point(0,2,0)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~3,3,~4),point(~2,5,~1)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~2,0,1),point(~1,2,4)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~2,~3,~3),point(~1,~1,0)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(2,~1,~1),point(3,1,2)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~2,~3,~4),point(~1,~1,~1)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(0,~1,~3),point(1,1,0)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~1,~2,~3),point(0,0,0)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~3,~2,~1),point(~2,0,2)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(~10,~8,~3),point(~9,~6,0)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(5,9,9),point(6,11,12)) ),
        world( box(point(1,3,2),point(4,7,8)), box(point(9,11,15),point(10,13,18)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(5,9,10),point(6,11,13)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(8,10,11),point(9,12,14)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(4,8,10),point(5,10,13)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(5,6,9),point(6,8,12)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(1,8,11),point(2,10,14)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(7,10,9),point(8,12,12)) ),
                world( box(point(1,3,2),point(4,7,8)), box(point(10,10,10),point(11,12,13)) )
]


val Validation_inputs = []

val Funs_to_use = [ "false", "true", "finished", "up", "down", "left", "right",
        "foreward", "backward", "<" ]

val Abstract_types = []

val Initial_program = NONE

val Reject_funs = []
fun restore_transform D = D

structure Grade : GRADE =
struct

type grade = int
val zero = 0
val op+ = Int.+
val comparisons = [ Int.compare ]
val toString = Int.toString
val pack = Int.toString
val unpack = fn S => case Int.fromString S of SOME N => N

val toRealOpt = NONE

val post_process = fn X => X

end
```

```
fun isInside( (Point as point(x,y,z), Box as box(
        Point1 as point(x1,y1,z1), Point2 as point(x2,y2,z2) )) : point * box ) : bool =
                (x1 <= x andalso x <= x2 andalso
                y1 <= y andalso y <= y2 andalso
                z1 <= z andalso z <= z2 )

fun output_eval_fun( I : int, _ : mainReturn, Y : mainReturn ) =
  case Y of mainReturn( World, Cnt ) =>
            case World of world( Box1, Box2 ) =>
                case Box2 of box( P1, P2 ) =>
                    if isInside(P1, Box1) andalso isInside(P2, Box1) then
                      { numCorrect = 1, numWrong = 0, grade = Cnt }
                           else
                      { numCorrect = 0, numWrong = 1, grade = 0 }

val Max_output_genus_card = 2
val Max_time_limit = 65536
val Time_limit_base = 2.0
```

Appendix C: ADATE Utilities

A series of utilities has been implemented for ADATE. This appendix gives a short description of some of them. They are included in the ADATE distribution.

c5conv

Since decision trees are a simple special case of functional programs, ADATE can generate them and in many cases produce better models than any other known machine learning technique if given enough time. C5.0 is the most well-known program for inducing decision trees and is extremely fast due to its mostly greedy heuristic. The C5.0 file format may be regarded as an unofficial standard in machine learning and is used by most specifications in the UCI Machine Learning Repository as well as many others.

The c5conv utility converts C5.0 files to an ADATE specification. It uses a .names file and a .data file in C5.0 format to automatically generate an ADATE specification file and is invoked in the following manner if these files are called hypothyroid.names and hypothyroid.data, for example.

```
[geirvatt@localhost ADATE]$ ./c5conv –f hypothyroid
```

This should leave the file hypothroid.spec in the current directory and a randomly reordered version of the .data file called hypothyroid.data.scrambled. The latter can be used for comparisons between ADATE and C5.0 with exactly the same data for training and validation/testing.

The first line in the .names file must contain the name of the target attribute followed by '.'. The only special keywords recognized in the .names file are 'continuous', meaning that the corresponding attribute has a floating point representation, and 'ignore', implying that the corresponding attribute and data will not be used. All floating point values are scaled to the range between -0.5 and +0.5. No comments are allowed in the .names file.

Note that the attribute names must be legal as identifiers in Standard ML. The values of an attribute should not include '?' in the .names file, which indicates a missing value in many machine learning data sets. The c5conv utility scans the .data file for columns that contain at least one '?' and then knows that there are missing values for the corresponding attribute without this being explicitly stated in the .names file. Thus, missing values are handled without the need to provide any special information in the .names file. If the .data file contains lines with type errors, that is attribute values not in agreement with the .names file, you will unfortunately get rather incomprehensible error messages from c5conv, which was not written to be user-friendly.

C5.0 automatically defines attributes values using the .data file whereas c5conv only allows the values given in the .names file, which gives a stricter check that the data file is correct.

Take a look at a generated .spec file to see how c5conv scales numerical values and handles data with missing values when translating C5.0 files to ADATE-ML.

ancestors.pl

This is a perl script that prints out all ancestors of a given individual, that is, can be employed to see all the intermediary forms in the evolutionary chain leading to the individual. Its usage can be found by invoking it with the --help option.

If it complains about “Too many missing children”, use the option “—tries 100” for example.