Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-249

MACE 2.0 Reference Manual and Guide

by

William McCune

Mathematics and Computer Science Division

Technical Memorandum No. 249

May 2001

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States Government and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

Contents

MACE 2.0 Reference Manual and Guide

William McCune

Abstract

MACE is a program that searches for finite models of first-order statements. The statement to be modeled is first translated to clauses, then to relational clauses; finally for the given domain size, the ground instances are constructed. A Davis-Putnam-Loveland-Logeman procedure decides the propositional problem, and any models found are translated to first-order models. MACE is a useful complement to the theorem prover Otter, with Otter searching for proofs and MACE looking for countermodels.

1 Introduction

MACE (Models And CounterExamples) is a program that searches for small finite models of first-order statements. It is frequently used along with our first-order theorem prover Otter [5, 4], with Otter searching for proofs and MACE looking for countermodels. The two programs accept almost the same language, so the same input file can usually be used for both programs.

MACE has been used for many applications including quasigroup existence problems [3], ortholattice problems [6], and lattice and Boolean algebra problems [7, 8]. Other successful programs that look for finite models of first-order statements are SEM [11] and FINDER [10]. A related class of programs can produce finite models when the search for a refutation fails. Examples of these are SATCHMO [2] and MGTP [1].

At its core, MACE has a Davis-Putnam-Loveland-Logeman propositional decision procedure named ANLDP. ANLDP can be used directly to decide propositional (SAT) problems given in conjunctive normal form (see Section 8). Section 6 gives differences between MACE 2.0 and previous versions. The MACE Web site is `http://www.mcs.anl.gov/AR/mace`.

2 A Little Motivation

Say you've just invented group theory by writing down the following three axioms,

$$
\begin{aligned}
e * x &= x \\
g(x) * x &= e \\
(x * y) * z &= x * (y * z)
\end{aligned}
$$

and you are wondering whether all groups are commutative. You prepare the following input file, named `group.in`.

```
set(auto).
list(usable).
  e * x = x.                      % left identity
  g(x) * x = e.                   % left inverse
  (x * y) * z = x * (y * z).      % associativity
  a * b != b * a.                 % denial of commutativity
end_of_list.
```

Now you give the input file to Otter to search for a proof (actually a refutation).

```
% otter < group.in > group.out
```

and to MACE to look for a countermodel (actually a model) of size 4, as follows.

```
% mace -n4 -p -c < group.in > group4.out
```

Both programs fail immediately. But looking at the Otter output makes you suspect that not all groups are commutative, so you go forward, looking for larger countermodels. The command

```
% mace -n6 -p -c < group.in > group6.out
```

succeeds, and the output file contains the following noncommutative group of order 6.

```
======================= Model #1 at 1.13 seconds:
e: 0
a: 1
b: 2
*:   | 0 1 2 3 4 5
   --+------------
   0 | 0 1 2 3 4 5
   1 | 1 0 3 2 5 4
   2 | 2 4 0 5 1 3
   3 | 3 5 1 4 0 2
   4 | 4 2 5 0 3 1
   5 | 5 3 4 1 2 0

g:     0 1 2 3 4 5
   ---------------
       0 1 2 4 3 5
```

Hmmm, very interesting: I wonder what happens if we add $x * x = e$ to our theory.

3 How to Tell MACE What to Do

Three kinds of input determine how MACE works. First, the clauses or formulas in the input file specify the theory for which you seek a model. Second, special commands in the input file put constraints on the models. Third, command-line options give general constraints on the search.

3.1 The Formulas

MACE reads the input (from stdin) and takes formulas and clauses from the lists `usable`, `sos`, `demodulators`, and `passive` as its basic theory.[1] Like Otter, MACE immediately transforms any first-order formulas to clauses.

MACE is a bit more restrictive than Otter in the language it accepts, and it interprets some symbols differently. See Section 4.

3.2 Constraints in the Input

Constraints are specified in an optional list `mace_constraints` in the input file.[2] (If you give Otter an input file containing a `mace_constraints` list, Otter ignores it.) Two kinds of constraint are accepted: assignments for the models and properties of relations or functions. Here is an example list that shows all of the types of constraint.

```
list(mace_constraints).
  assign(e, 0).                   % constant symbol
  assign(g(2), 1).                % function symbol
  assign(3*4, 2).                 % function symbol
  assign(P(1), T).                % relation symbol
  assign(Q(0,3), F).              % relation symbol
  property(same(_,_), equality).
  property(lt(_,_), order).
  property(g(_), bijection).
  property(_*_, quasigroup).
end_of_list.
```

The assignments simply give function values or relation values for particular members of the domain.[3] Members of the domain are always named $0, 1, \ldots, n-1$, where n is the domain size. The Boolean constants (relation values) are named `T` and `F`. Note that assigning values to constants can also be done with the `-c` command-line option (see the next subsection). The following properties of function and relation symbols can be specified in the `mace_constraints` list.

`equality`
: This applies to binary relation symbols. It is necessary only if a nonstandard equality symbol is being used, because any binary relation recognized by Otter as an equality symbol is also recognized by MACE as an equality symbol. See Section 4.2.

`order`
: This applies to binary relation symbols. It is necessary only if a nonstandard order symbol is being used. MACE (but not Otter) automatically recognizes binary $<$ as an order relation. The "order"

[1] One can argue that the hot list should also be considered as part of the basic theory, because Otter uses the hot list to make inferences. MACE ignores the hot list, however, because hot list clauses almost always occur also in usable or sos, and MACE suffers if it gets duplicate clauses. I suppose MACE could get around this by doing a subsumption check.

[2] Previous versions of MACE used the `passive` list for constraints.

[3] Why not place assignments in with the clauses that specify the theory? This can be done, but such assignments might not make sense if the input is also being used for Otter.

is the obvious order on the members of the domain: $0 < 1 < \ldots < n - 1$. See the example input files `ordered_semi.in` and `cd.in` included in the MACE distribution package.

`bijection`
This applies to unary function symbols. The list of function values is a permutation of the domain.

`quasigroup`
This applies to binary function symbols. If you write down the table for a finite quasigroup, each row and each column is a permutation of the domain.

3.3 Command-Line Options

`-n` n This gives the starting domain size for the search. The default value is 2. If you also give an `-N` option, MACE will iterate domain sizes up through the `-N` value. Otherwise, MACE will search only for the `-n` value. For example,

Options	Search
`-n4`	4
`-N6`	2,3,4,5,6
`-n4 -N6`	4,5,6

`-N` n This gives the ending domain size for the search. The default is the value of the `-n` option.

`-c` This says that constants in the input should be assigned unique elements of the domain. If the number of constants in the input is greater than the domain size n, the first n constants are given values, and the rest are unconstrained. This is a useful option because it eliminates lots of isomorphism from the search. But it can block all models, especially when used with other constraints.

`-p` (Lower case.) This option tells MACE to print models in a nice tabular form as they are found. This format is meant for human consumption.

`-P` (Upper case.) This option tells MACE to print models in an easily parsable form. This format has an Otter-like syntax and can be read by most Prolog systems.

`-I` This option tells MACE to print models in IVY form. This format is a Lisp S-expression and is meant to be read by IVY [9], our proof and model checker.

`-m` n This tells MACE to stop after finding n models. The default is 1.

`-t` n This tells MACE to stop after about n seconds. The default is unlimited. *MACE ignores any `assign(max_seconds, n)` commands that might be in the input file. Such commands are used by Otter only.*

`-k` n This tells MACE to stop if it tries to allocate more than n kilobytes ofmemory. The default is 48000 (about 48 megabytes). *MACE ignores any `assign(max_mem, n)` commands that might be in the input file. Such commands are used by Otter only.*

`-x` This is a special-purpose constraint designed to reduce isomorphism in quasigroup problems. It applies only to binary function `f`. See [3].

`-h` This tells MACE to print a summary of these command-line options.

4 Language Accepted by MACE

MACE accepts nearly the same input as Otter. First we list the main differences from Otter; then we give a short review of Otter's language.

4.1 Differences from Otter's Language

1. MACE does not accept function symbols with arity greater than three or relation symbols with arity greater than four.
2. MACE does not allow symbols with different arities, for example, `f(f,x)`.
3. MACE does not allow a symbol to be used as both a relation symbol and a function symbol.
4. MACE ignores answer literals. In fact, MACE removes all answer literals before it starts looking for models.
5. The natural numbers `0,1,2,...` are ordinary constants to Otter, but they have special meanings to MACE. In particular, MACE interprets them as elements of the domain. If you ask MACE to look for a model of size n, and there are constants $\geq n$ in the input, MACE will get confused and quit with an error message.
6. On the other hand, the evaluable ("dollar") functions and relations, for example `$SUM` and `$LT`, have special meanings to Otter, but they are treated by MACE as ordinary symbols. As a result, an input file containing evaluable symbols can produce both a refutation with Otter and a model with MACE. Here is an example.

```
set(hyper_res).
list(sos).
  -P(x) | P($SUM(x,x)).
  P(1).
  -P(2).
end_of_list.
```

4.2 A Quick Review of the Language Otter Accepts

See the Otter manual [5] for a thorough description of the language.

Clauses vs. Formulas. You can use either clauses or formulas. (Most people use clauses. If you use formulas, they are immediately translated to clauses.) Here are some corresponding examples.

```
list(usable).            % clauses
  -P(x) | -Q(x) | R(x).
  -P(x) | -Q(x) | S(x).
  f(e,x) = x.
  f(g(x),x) = e.
end_of_list.
```

```
formula_list(usable).   % formulas
  all x (P(x) & Q(x) -> R(x) & S(x)).
  exists e ((all x (f(e,x) = x)) &
            (all x exists y (f(y,x) = e))).
end_of_list.
```

Variables vs. Constants in Clauses. Clauses do not have explicit quantifiers, so we need a rule to distinguish variables from constants. The default rule is that symbols starting with `u` through `z` are variables. If the command `set(prolog_style_variables)` is in effect, symbols starting with upper-case letters are variables.

Equality Symbols. How do we recognize binary relations as equality relations? The default rule is that the symbol = and symbols matching the pattern [`Ee`][`Qq`].* are equality symbols. If the input contains the command `set(tptp_eq)`, then `equal` is the one and only equality symbol.

Infix Notation. One can declare binary symbols to be infix and to have a precedence and associativity so that some parentheses can be omitted. Many symbols such as = and `*` have built-in declarations.

5 How MACE Works

The methods used by MACE are described in detail in [3]. Here is a summary.

For a given domain size, MACE transforms the (first-order) input into an equivalent propositional problem. This is possible because, for a fixed finite domain, the first-order problem is decidable. The propositional problem is then given to a DPLL (Davis-Putnam-Loveland-Logeman) procedure. If satisfiability is detected, the propositional model is transformed into a first-order model of the original problem.

Consider the following input file.

```
list(usable).
  even(a).
  -even(x) | even(s(s(x))).
  -even(s(a)).
end_of_list.
```

MACE first flattens the clauses into a relational form. This step involves replacing each n-ary function with an $n+1$-ary relation. MACE's output for this example contains something like

```
Processing clause: -a(v0) | even(v0).
Processing clause: -s(v0,v1) | -s(v1,v2) | -even(v0) | even(v2).
Processing clause: -a(v0) | -s(v0,v1) | -even(v1).
```

If we ask for models of size 3, MACE generates propositional clauses corresponding to all instances of the transformed clauses over the set $\{0,1,2\}$. The output also contains the statements

```
Function s/2 well-defined and closed.
Function a/1 well-defined and closed.
```

which indicate that MACE has generated propositional clauses asserting that the new $n+1$-ary relations are functions. The DPLL procedure finds a model of the set of propositional clauses, and the propositional model is transformed into the following first-order model.

```
a: 2        even:  0 1 2          s:      0 1 2
                ---------              ---------
                   T F T                  0 0 1
```

Scalability. Unfortunately, this method does not scale well as the domain increases or as the size of clauses increases. Consider a distributivity axiom, $x * (y + z) = (x + y) * (x + z)$. The transformation to relational form produces the following two clauses.

```
-+(v0,v1,v2) -+(v0,v3,v4) -*(v4,v2,v5) -+(v3,v1,v6) *(v0,v6,v5)
-+(v0,v1,v2) -+(v0,v3,v4) *(v4,v2,v5) -+(v3,v1,v6) -*(v0,v6,v5)
```

For a domain of 6, each of these (7-variable) clauses produces $6^7 = 279,936$ propositional clauses. MACE can usually handle this many clauses, but it's hard to fight exponential behavior. The program SEM [11] is usually better than MACE for large clauses or large domains.

6 Differences from Previous Versions

Major changes from earlier versions of MACE are listed here.

1. Previous versions of MACE called Otter to parse the input and to produce an intermediate form that was given to a program named ANLDP. MACE 2.0 is self-contained, making it easier to install and run.
2. Previous versions of MACE worked for a fixed domain size, and there was a separate script (mace-loop) to iterate through domain sizes and calling MACE.
3. Previous versions of MACE used Otter's `passive` list for constraints (assignments and properties). MACE 2.0 uses the new list `mace_constraints` for that purpose; clauses in `passive` are now taken as part of the theory.
4. MACE 2.0 allows answer literals in the clauses. (Answer literals are removed by MACE before the search for models.)
5. Previous versions of MACE could handle sorted logic (with disjoint domains). MACE 2.0 cannot. Most of the code for sorted logic is still in place, so it is possible that future versions will handle sorted logic.

   Sorted logic can sharply cut down the search time. Consider a domain of size 12 that can be partitioned into 8 and 4. A 2-variable relational clause, with one variable for each sort, produces 144 propositional clauses with unsorted logic and 32 clauses in the sorted case. Let us know if you need sorted logic.

6. Previous versions of MACE had a checkpointing feature whereby the state of the search was periodically backed up to a file, and the search could be resumed from one of those states. MACE 2.0 does not have this feature.

7 Calling MACE From Other Programs

MACE returns an exit code when it terminates. This makes it convenient to call MACE from other programs. Here is a list of MACE's exit codes. (This list changes from time to time; the current list can be found in the source file `Mace.h`.)

11 `(ABEND_EXIT)` This usually indicates an error in the input (not all input errors are covered by `INPUT_ERROR_EXIT` below). Occasionally it is caused by a bug in MACE. When you get this exit code, look in the output for an error message.

12 `(UNSATISFIABLE_EXIT)` MACE completed its search and determined that no models exist within the given domain size(s) and other constraints. *It does not mean that the input clauses are unsatisfiable.*

13 `(MAX_SECONDS_EXIT)` MACE terminated because of the time limit given on command line (with `-t`).

14 `(MAX_MEM_EXIT)` MACE terminated because of the memory limit given on the command line (with `-k`).

15 `(MAX_MODELS_EXIT)` MACE terminated because it found the number of models requested on the command line (with `-m`).

16 `(ALL_MODELS_EXIT)` MACE completed its search and found all models (at least one) within the given constraints.

17 `(SIGINT_EXIT)` MACE terminated because it received the interrupt signal.

18 `(SEGV_EXIT)` MACE crashed.

19 `(INPUT_ERROR_EXIT)` Errors were found in the input. The output file should point to the error(s).

Say we have a list of equations containing a binary function symbol `f`, and we wish to remove the equations that have a noncommutative model of size ≤ 4. If we put the equations in a file, with one equation on each line, for example,

```
f(f(x,f(f(z,x),x)),f(z,f(y,x))) = z.
f(f(f(x,f(z,x)),x),f(z,f(y,x))) = z.
f(f(f(f(y,x),z),x),f(f(u,y),x)) = x.
f(f(f(f(y,x),z),x),f(f(y,u),x)) = x.
```

we can write a simple program to loop through the equations, calling MACE for each and printing those that have no noncommutative models of size ≤ 4. Here is an example Perl program that does the job.

```
#!/usr/local/bin/perl5

$mace = "/home/mccune/bin-linux/mace"; # MACE binary
$unsatisfiable_exit = 12;  # exit code of interest
$input = "/tmp/mace$$";    # temporary input file

while ($equation = <STDIN>) {
    open(FH, ">$input") || die "Cannot open file $input";
    print FH "list(usable). $equation f(0,1)!=f(1,0). end_of_list.\n";
    close(FH);
    $rc = system("$mace -N4 < $input > /dev/null 2> /dev/null");
    $rc = $rc / 256;    # This gets the actual exit code.
    if ($rc == $unsatisfiable_exit) { print $equation; }
}
system("/bin/rm $input");
```

If our data file is named `identities` and our Perl script is named `commute4_filter`, then the command

```
% commute4_filter < identities > candidates
```

will remove two of the four equations within a few seconds.

8 The ANLDP Propositional Decision Procedure

If you have a propositional (SAT) problem in conjunctive normal form, you can call MACE's DPLL procedure directly with the program ANLDP. ANLDP is included in the MACE distribution package.

Input to ANLDP is a sequence of integers (no comments are allowed). The propositional variables are `1,2,3,...`. Positive integers are positive literals, negative integers are negative literals, and 0 marks the ends of clauses. For example, here is an (unsatisfiable) input consisting of four 2-literal clauses.

```
1 2 0
1 -2 0
-1 2 0
-1 -2 0
```

The command-line options of ANLDP are a subset of MACE's:

- `-p` (Lower case.) This tells ANLDP to print models as they are found.
- `-m` n This tells ANLDP to stop after finding n models. The default is 1.
- `-t` n This tells ANLDP to stop after about n seconds. The default is unlimited.
- `-k` n This tells ANLDP to stop if it tries to allocate more than n kilobytes of memory. The default is 48000 (about 48 megabytes).

`-s` This tells ANLDP to perform unit subsumption as it searches. (Unit subsumption is always performed on the input.) When ANLDP gets a new unit (by splitting or by unit propagation), two operations are ordinarily performed: (1) unit resolution, to remove complementary literals from all clauses, and (2) unit subsumption, to mark as subsumed all clauses containing the unit as a literal. Because of our data structures, unit subsumption nearly always costs more time than it saves. But this option allows you to use unit subsumption if you wish.

ANLDP is an implementation of the Davis-Putnam-Loveland-Logeman procedure. Efficient data structures and algorithms are used, but the procedure is otherwise standard. When the time comes to select the next propositional variable for splitting, ANLDP simply takes the first variable of the first shortest positive clause. Details of the implementation can be found in [3].

References

[1] H. Fujita and R. Hasegawa. A model generateion theorem prover in KL1 using a ramified-stack algorithm. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 535–548, 1991.

[2] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction, Lecture Notes in Computer Science, Vol. 310*, pages 415–434, Berlin, 1988. Springer-Verlag.

[3] W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Tech. Report ANL/MCS-TM-194, Argonne National Laboratory, Argonne, IL, May 1994.

[4] W. McCune. Otter. `http://www.mcs.anl.gov/AR/otter/`, 1994.

[5] W. McCune. Otter 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994.

[6] W. McCune. Automatic proofs and counterexamples for some ortholattice identities. *Information Processing Letters*, 65:285–291, 1998.

[7] W. McCune and R. Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*, volume 1095 of *Lecture Notes in Computer Science (AI subseries)*. Springer-Verlag, Berlin, 1996.

[8] W. McCune and R. Padmanabhan. Automated deduction in equational logic and cubic curves. `http://www.mcs.anl.gov/home/mccune/ar/monograph/`, 1996.

[9] W. McCune and O. Shumsky. IVY: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios, and J Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 16. Kluwer Academic, 2000.

[10] J. Slaney. FINDER, finite domain enumerator: Version 1.0 notes and guide. Tech. Report TR-ARP-10/91, Automated Reasoning Project, Australian National University, Canberra, Australia, 1991.

[11] J. Zhang and H. Zhang. SEM: A system for enumerating models. In *Proceedings of the International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995.