MCheck :
A Model Checker for LTL and CTL Formulas

Jeff Sember
CMPT 880
Spring 2005

Contents

Chapter 1

Introduction

MCheck is designed to be a simple text-based interpreter that can be used as an instruction aid for the study of temporal logics.

The program processes *models* of a system and *specifications* or *properties* which a system is assumed to have. Its main function is to determine whether a model satisfies a set of specifications. The program can also test specification formulas (of a particular type) for equivalence, and can convert formulas to equivalent, 'reduced' forms.

1.1 Kripke Models

A model is represented as a Kripke model, which is a 3-tuple $(S, \longrightarrow, L)$, where S is a finite set of states, $\longrightarrow \subset S \times S$ is a transition relation, and $L : S \rightarrow \mathbb{P}(\text{Atoms})$ is a labelling function that associates a subset of propositional variables (Atoms) with each state in S.

MCheck actually uses an augmented form of Kripke model: $(S, \longrightarrow, L, I)$, where $I \subset S$ is a set of *initial states*.

1.2 Temporal Logic

The program supports three flavors of temporal logic formulas: LTL, Linear Time Temporal Logic; CTL, Computation Tree Logic; and CTL*, which is a superclass of LTL and CTL. Formulas can be specified in any of these three variants, but only LTL and CTL formulas can be verified by MCheck.

Chapter 2

MCheck in Operation

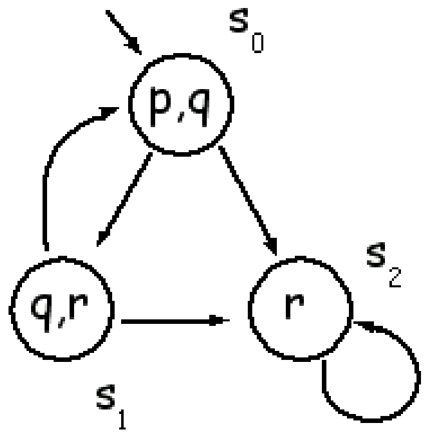

Figure 2.1: A representation of a Kripke model (from [3]).

The basic function of MCheck is to examine a Kripke model and see if it satisfies one or more temporal logic formulas. Suppose the Kripke model is that of figure 2.1. This model could appear in an MCheck input file in the form shown in figure 2.2.

A model is surrounded by curly braces, { and }. Within the braces are a number of state definitions, which consist of a state number, a nonempty list of states the state has transitions to, and a list of the propositional variables that are true in that state.

The '>' preceding the first state indicates that it is one of (in fact, the only) initial states of the model. If no states are marked '>', then every state is considered

```
{ > 0   1 2    p q
    1   0 2    q r
    2   2      r
}
```

Figure 2.2: MCheck source for Kripke model of figure 2.1.

a start state.

While it is natural to list states and transitions in increasing order, this is not required. The state numbers don't have to start at zero or be contiguous. The model of figure 2.1 could be represented in the form shown in figure 2.3 as well.

```
{75 2000 300 r q >300 75 2000 p q 2000 2000 r}
```

Figure 2.3: Another representation of the Kripke model.

The states have been renamed $0 \rightarrow 300, 1 \rightarrow 75, 2 \rightarrow 2000$, and the line breaks have been removed.

2.1 Checking models against specifications

To test whether the model satisfies some specifications, the specifications are listed following the model; see figure 2.4. The program treats portions of lines beginning with `--`, `//`, or `%` as comments, so these can be omitted.

Suppose the text of figure 2.4 was saved as `test.txt`. To verify the model, the program is run by typing:

```
mch test.txt
```

The program will respond with the output shown in figure 2.5. Each of the three LTL formulas are tested in the model, starting at state #0 (since it is the only initial state). The first two are satisfied, and the third is not. A counterexample is given, which is a sequence of states that violate the property. In this case, the infinite sequence 0 1 0 1... never reaches state #2, where r is always true.

```
-- model
{ > 0   1 2    p q
    1   0 2    q r
    2   2      r
}

-- specifications
p U r   -- LTL: p until r
F r     -- LTL: future r
F G r   -- LTL: eventually always r
```

Figure 2.4: Model and specifications

```
Parsed model, 3 states

LTL : p U r

Satisfied.

LTL : F r

Satisfied.

LTL : F G r

Not satisfied; counterexample:
  {0 1 0 1}*
```

Figure 2.5: MCheck output

2.2 Splitting source files into pieces

It is often convenient to put the model to be tested in one file, and have the formulas to be tested against in another. Suppose the model of figure 2.2 is saved as `model.txt`, and the set of CTL formulas (some are taken from [3]) of figure 2.6 is saved as `formulas.txt`. Verifying the model is then done by typing:

```
mch model.txt formulas.txt
```

```
p & q
!r
T
EX(q & r)
!EF(p & r)
AF r
E[(p & q) U r]
AG(p | q | r -> EF EG r)
AG(F (r | p))
```

Figure 2.6: CTL formulas

The program will respond with the output shown in figure 2.7. Notice that the first three formulas are both CTL and LTL formulas. The program displays the type of formula with the output, as one of `C/L`, `LTL`, `CTL`, or `CTL*`. Notice that the last formula cannot be verified by the MCheck program, since it is of type CTL*.

If the input files contain a model but no specifications, MCheck simply displays a description of the model. If the input files contain specification formulas but no model, the formulas are displayed. If the input files contain more than one model, the new model will replace the old at that point. Thus several sequences of model + specifications can be concatenated together during a single program execution.

2.3 Testing equivalence of LTL formulas

The MCheck program can determine if two LTL formulas are equivalent. Two LTL formulas are paired, preceded with `?` and `:` (see figure 2.8).

```
Parsed model, 3 states

C/L : p & q

Satisfied.

C/L : !r

Satisfied.

C/L : T

Satisfied.

CTL : EX (q & r)

Satisfied.

CTL : !EF (p & r)

Satisfied.

CTL : AF r

Satisfied.

CTL : E[(p & q) U r]

Satisfied.

CTL : AG ((p | q) | r -> EF EG r)

Satisfied.

CTL*: AG F (r | p)

(cannot check mixed CTL/LTL formula...)
```

Figure 2.7: Model tested against CTL specifications

```
? p R q
: q W (p & q)

? !(p U q)
: (!p | (q & (!q | !p))) R !q

? G F (p | q)
: ! F G !(p & q)
```

Figure 2.8: LTL formula pairs

If these formulas are saved as `ltlequiv.txt`, then testing their equivalence is done by typing:

```
mch ltlequiv.txt
```

The program will respond with the output shown in figure 2.9. If the formulas are not equivalent, a sequence of *state formulas* is displayed representing a path through a model that satisfies one of the formulas but not the other. Each state formula describes the restrictions that exist in that state for the values of the propositional variables. This indicates that there is a sequence of propositional variable truth assignments that satisfies the sequence of state formulas, and thus satisfies one of the input formulas, while failing to satisfy the other input formula.

2.4 Special options

During the processing of a CTL or LTL formula, the MCheck program may *reduce* the formula into a simpler (but often longer) form. To see what the converted formulas look like, the -r option is used. For example, the formulas of figure 2.6 can be examined by typing:

```
mch -r formulas.txt
```

Part of the output is shown in figure 2.10.

When the MCheck program parses a formula, it removes all the parentheses, since once a parse tree is formed they are not needed. When displaying a parsed formula, it avoids inserting extraneous sets of parentheses. It may be desirable in

```
Comparing: p R q

     with: q W (p & q)

Equivalent.

Comparing: !(p U q)

     with: (!p | (q & (!q | !p))) R !q

Equivalent.

Comparing: G F (p | q)

     with: !F G !(p & q)

Not equivalent.

 first allows: T T {T T T T p T p p q T}*
```

Figure 2.9: Testing equivalence of LTL formulas

```
C/L : p & q
      p & q

CTL : EX (q & r)
      EX (q & r)

CTL : !EF (p & r)
      !E[!B U (p & r)]

CTL : AG ((p | q) | r -> EF EG r)
      !E[!B U (!((!p & !q) & !r) & !E[!B U !AF !r])]

CTL*: AG F (r | p)
      AG F (r | p)
```

Figure 2.10: Reduced formulas

some cases to see what a fully parenthesized formula looks like. This is done by using the -p option:

```
mch -p formulas.txt
```

Part of the output is shown in figure 2.11.

When LTL formulas are manipulated, either for testing model satisfaction or formula equivalence, it is possible to have MCheck display the *Büchi automata* (see section 3.3) generated during the operation. The `-b` command line argument enables this feature; see Appendix A.

```
C/L : ((p) & (q))

CTL : (EX ((q) & (r)))

CTL : (!(EF ((p) & (r))))

CTL : (AG ((((p) | (q)) | (r)) -> (EF (EG (r)))))

CTL*: (AG (F ((r) | (p))))
```

Figure 2.11: Fully parenthesized formulas

Chapter 3

Algorithms and Data Structures

3.1 Data structure hierarchy

Figure 3.1 provides an informal view of the relationship between the various data structures which the MCheck program uses.

The main program uses the Scanner class to parse input files, which in turn uses the DFA class (Deterministic Finite State Automaton) to tokenize the input files.

The main program calls on the Model and Formula classes to construct Kripke models and temporal logic formulas. These are then passed to the appropriate model checking class, CTLCheck or LTLCheck, for analysis.

The Scanner, Model, and Formula classes use the Vars class to maintain a symbol table of propositional variable names. The Vars class uses a hash table to store these names.

The Formula class stores each formula as a tree, which is contained within the Forest class. The underlying data structure for these trees is a directed graph, which is implemented in the Graph class.

3.2 CTL model checking algorithm

The algorithm used to check models against CTL formulas is the labelling algorithm given in section 3.6.1 of [3]. Since formulas are parsed into tree structures, it is a simple matter to build a list of subformulas by traversing a formula's tree.

The subformulas are sorted into reverse order of tree depth, so that the lowest nodes of the tree are processed first. Every state in the model is examined for each subformula, to see if it should be *labelled* with the subformula. Some of these

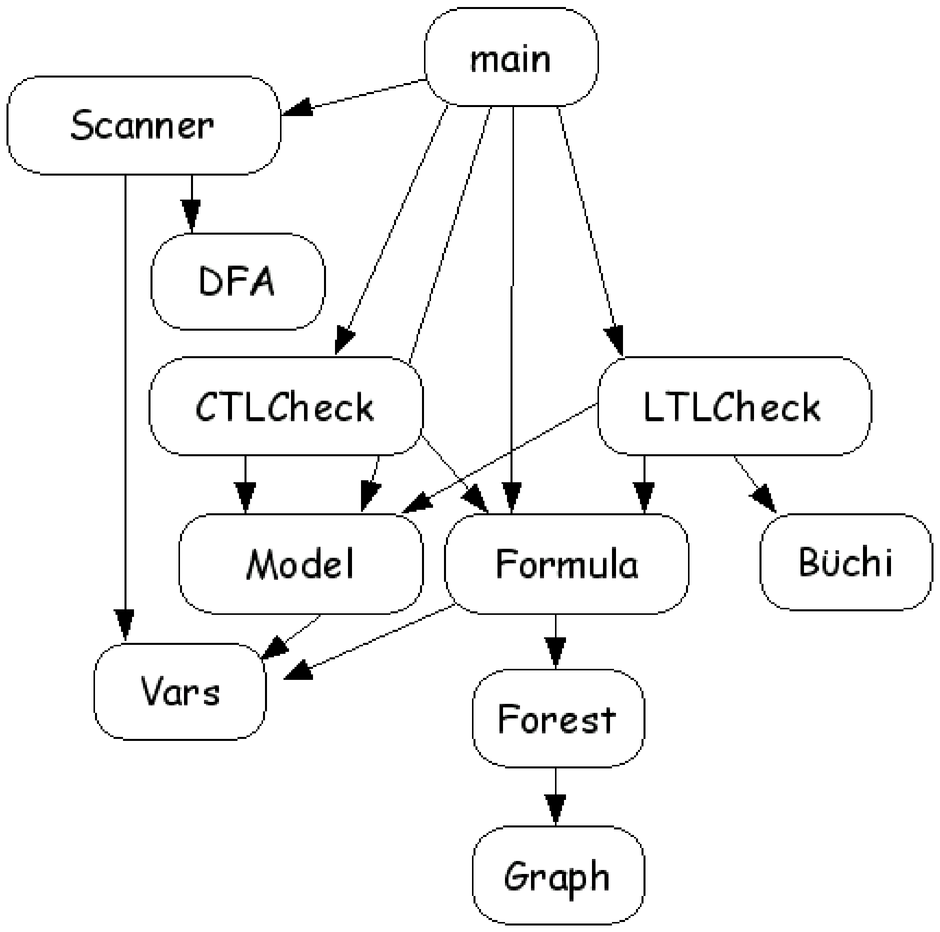

Figure 3.1: Data structure hierarchy

labelling procedures are repeated until a fixed point is reached (in other words, the model is scanned again until no changes have been made).

When the labelling algorithm is complete, the initial states of the model are examined to see if they have been labelled with the root subformula. If all such states have been so labelled, the model satisfies the CTL formula.

3.3 LTL model checking algorithm

LTL model checking is performed by converting both the Kripke model and the *negation* of the LTL formula to a *Büchi automaton*, a form of finite automaton. A third automaton is constructed as the product of these two. Model verification is performed by determining if this product automaton recognizes any infinite strings. The use of automata to perform model checking is presented in [1], and the LTL-to-Büchi automaton conversion algorithm is found in [2].

Equivalence testing of LTL formulas F_1 and F_2 is also achieved using Büchi automata. Automatons B_1 and B_2 are constructed from F_1 and $\neg F_2$; if the *language* of the product of these two automatons is not empty, then there exist models that satisfy F_1 but not F_2. Similarly, automatons B_1' and B_2' are constructed from $\neg F_1$ and F_2, to see if there exist models that satisfy F_2 but not F_1. If the languages of both product automatons are empty, then the two LTL formulas are equivalent.

Converting a Kripke model to a Büchi automaton is quite simple. Converting an LTL formula is somewhat involved, and the size of the resulting Büchi automaton can be exponential in the size of the formula. In addition, constructing the product of two automatons yields an automaton that has a state count that is the product of the original state counts.

3.4 Formula manipulations

Both the CTL and LTL model checking classes manipulate input formulas into forms more appropriate for their respective algorithms. These manipulations include formula *rewriting*, in which subformulas rooted at nodes containing particular logical connectives (operators) are replaced with equivalent but structurally different subformulas which use a smaller set of connectives. The Formula class uses a simple scripting language to perform this rewriting process in a fairly painless manner. Since the LTL and CTL classes use different *adequate sets* of connectives, each class performs the formula rewriting by referencing a different table of rewriting scripts. Additional scripts are employed to remove ‘double negation’ sequences, and (for LTL checking) to push negations inside, or lower, within the formula tree.

Formulas are also converted to directed acyclic graphs (DAGs). This $O(n^2)$ algorithm (where n is the number of nodes in the formula) merges identical subformulas, which has the effect of changing the tree to a directed acyclic graph. This yields two benefits: first, the number of nodes in the formula is reduced; and second, subformulas can be tested for equality by simply comparing their node numbers, instead of requiring a traversal of the two subformula trees.

Chapter 4

Future Development

There are a number of possible ways that the MCheck program could be improved.

- The counterexamples displayed by MCheck for LTL formulas which a model fails to satisfy are often lengthy, with many runs of repeated states. This is clearly an area where some sort of optimization would be helpful.
- When two LTL formulas are found to not be equivalent, the program displays a sequence of formulas that can produce a sequence of propositional variable truth assignments that satisfy exactly one of the two input formulas. Such a set of truth assignments is not explicitly given, however. This would be a useful feature to have; this would essentially entail constructing a Kripke model that satisfies one of the formulas but not the other.
- CTL formula equivalence testing is not supported. The author does not know whether an algorithm exists for this.
- Fairness constraints could be added. For LTL formulas at least, this would be relatively straightforward; the Büchi automata class could be easily modified to support this feature.
- Satisfaction testing for CTL* formulas could be added, since it mainly entails combining the LTL satisfaction testing as a subroutine to be used by the CTL labelling algorithm; details can be found in [1].

Appendix A

Running the Program

The MCheck executable is named `mch`, and is designed to be run from a UNIX (or DOS) command line. There are two methods for doing this:

(i) Specifying the input files as a command line argument:

```
mch <opts> <file1> <file2> ...
```

(ii) Piping the input files into the program as 'standard input'. In UNIX, this looks like:

```
cat <file1> <file2> ... | mch <opts>
```

For DOS, the 'type' command is probably desired:

```
type <file1> <file2> ... | mch <opts>
```

Options are represented by `<opts>`. These are zero or more single-letter commands, each preceded by `-`. Options may be grouped together; i.e. `-er`. See table A.1 for a list of valid options.

The input files are denoted `<file`*x*`>`. An input file is a text file that contains zero or more model definitions, LTL formulas, and CTL formulas.

The format of an input file is shown in table A.2. Some details to note include:

- Any text from `--`, `//`, or `%` is treated as a comment, and is ignored to the end of the current line.
- `x`* means zero or more `x`'s.
- `x`$^+$ means one or more `x`'s.
- [`x`] means zero or one `x`'s.

Argument	Function
-b	Displays Büchi automata (for LTL formulas only)
-e	Causes input files to be echoed to the screen as they are processed
-h	Prints the help message
-m	Shows formulas as they are ‘marked’ (for CTL formulas only)
-p	When displaying formulas, suppresses filtering of extraneous parentheses
-r	Displays formulas after reduction to minimally adequate set of connectives
-v	Verbose mode: prints extra information during CTL, LTL analysis

Table A.1: Options

- $\langle$num$\rangle$ is an integer ≥ 0.
- $\langle$propvar$\rangle$ is a propositional variable, which is any sequence of letters, digits, or underscore, that does not begin with a digit. Note that some sequences form *reserved words* such as `F` and `AG` which cannot be used as propositional variables. Note that no reserved words start with a lower-case letter.
- `T` and `B` refer to `True` and `Bottom` (or `False`). These are often represented in the literature as $\top$ and $\bot$.
- `!` represents negation ($\neg$ in the literature).
- Every $\langle$state_def$\rangle$ must contain at least one $\langle$propvar$\rangle$. The special variable _ can be used as a placeholder to mean ‘no variables true’.
- All operators are left-associative, except for: `->`, `U`, `R`, `W`, `A[..U..]`, and `E[..U..]`.
- The precedence of the formulas can be found in [3].

```
⟨file⟩ ::= ⟨command⟩*

⟨command⟩ ::= ⟨model⟩ | ⟨form⟩ | ? ⟨form⟩ : ⟨form⟩

⟨model⟩ ::= { ⟨state_def⟩* }

⟨state_def⟩ ::= [ > ] ⟨num⟩ ⟨num⟩* ⟨propvars⟩

⟨propvars⟩ ::= ( ⟨propvar⟩ )+ | _

⟨form⟩ ::= ⟨propvar⟩ | T | B
     | ( ⟨form⟩ )
     | ! ⟨form⟩ | ⟨form⟩ -> ⟨form⟩
     | ⟨form⟩ & ⟨form⟩ | ⟨form⟩ | ⟨form⟩
     | ⟨form⟩ U ⟨form⟩ | ⟨form⟩ R ⟨form⟩ | ⟨form⟩ W ⟨form⟩
     | X ⟨form⟩ | F ⟨form⟩ | G ⟨form⟩
     | AG ⟨form⟩ | EG ⟨form⟩
     | AF ⟨form⟩ | EF ⟨form⟩
     | AX ⟨form⟩ | EX ⟨form⟩
     | A[ ⟨form⟩ U ⟨form⟩ ]
     | E[ ⟨form⟩ U ⟨form⟩ ]
```

Table A.2: EBNF Grammar for Input Files

Appendix B

Example Files

A summary of the files included in this distribution of MCheck is given in table B.1. These include files discussed in this document, as well as some additional examples.

File	Description
`elevator.smv`	NuSMV program from which `elevator.txt` was derived
`elevator.txt`	Simple elevator problem, from assignment #3
`formulas.txt`	Formulas from figure 2.6
`ltlequiv.txt`	Formulas from figure 2.8
`mch`	Linux executable
`mch.exe`	Windows executable
`mch.pdf`	This document
`model.txt`	Model from figure 2.2
`model2.txt`	Model #2, for test purposes (see figure B.1)
`mutex.txt`	Mutual Exclusion problem, from Figure 3.11 of [3]
`sat2.txt`	A set of formulas satisfied by model #2
`unsat2.txt`	A set of formulas not satisfied by model #2

Table B.1: Sample source files

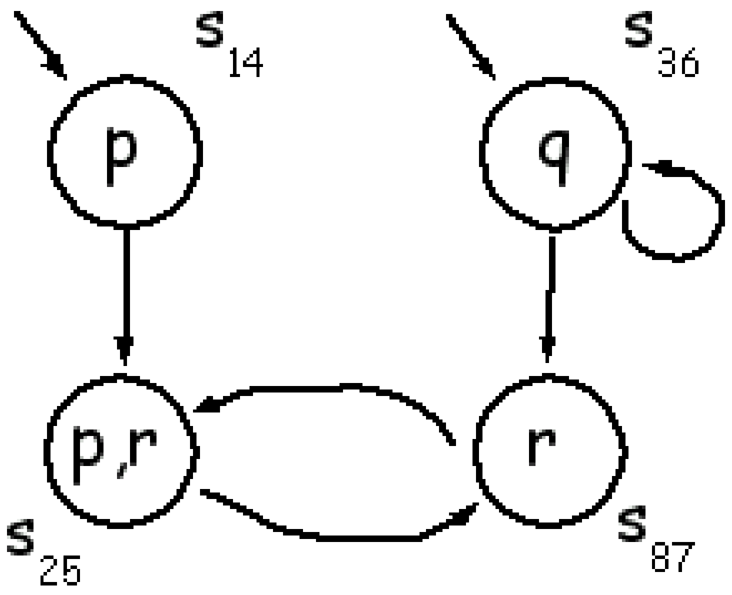

Figure B.1: Model #2

Bibliography

[1] E. M. Clarke, O. Grumberg, D. Peled. 1999. Model Checking. MIT Press.

[2] R. Gerth, D. Peled, M.Y. Vardi, P. Wolper. 1995. Simple On-the-fly Automatic Verification of Linear Temporal Logic. *Proceeding of 15th Workshop Protocol Specification, Testing, and Verification*, 3–18.

[3] M. Huth and M. Ryan. 2004. Logic in Computer Science: Modelling and Reasoning about Systems, 2nd Edition. Cambridge.