Memorax

User Manual

Carl Leonardsson

October 22, 2012

Copyright (C) 2012 Carl Leonardsson
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ''GNU
Free Documentation License''.

Contents

1 Introduction **3**

2 Contact / Bug Report **4**

3 Installation **4**
- 3.1 Requirements . . . 4
- 3.2 Basic Installation . . . 4
- 3.3 Installation Options . . . 4
- 3.4 Troubleshooting . . . 5

4 Usage **5**
- 4.1 Using the Command Line Interface . . . 6
- 4.2 Using the Graphical Interface . . . 8

5 Tutorial **9**

6 The Rmm language **12**
- 6.1 Machine Model and Memory Addressing . . . 13
- 6.2 An Example . . . 15
- 6.3 Semantics . . . 17
  - 6.3.1 Statements . . . 17
  - 6.3.2 Arithmetical and Boolean Expressions . . . 17
  - 6.3.3 Instructions Informally . . . 17
  - 6.3.4 Control Structures . . . 20
  - 6.3.5 TSO Semantics . . . 22
- 6.4 Grammar . . . 26
- 6.5 Macros . . . 28

7 Abstractions **29**
- 7.1 PB - Bounded Buffers with Predicate Abstraction . . . 29
- 7.2 SB - Single Buffer . . . 30

References **30**

GNU Free Documentation License **31**

1 Introduction

Memorax is a tool for verification of safety properties in programs running under relaxed memory models, and for automatic inference of memory fences that are necessary in order to guarantee satisfaction of those safety properties.

The verification is achieved through state space exploration and specifically control state reachability analysis. Inference of fences is achieved by repeated reachability analysis according to the method described in [3].

Programs running under relaxed memory models tend to have an infinite state space, even in the cases where the same program would have a finite state space if executed under SC. For this reason, explicit state space analysis is impossible. The problem can be handled by using under approximation, over

approximation or by using exact, infinite state analysis methods such as the well-quasi ordering method [1]. Memorax is built to accommodate different approaches.

Currently Memorax supports two approaches for verification/fence insertion of programs executed under the TSO memory model. Details are given in section 7.

This manual will assume basic knowledge about memory models. An introduction to the topic can be found in [4].

2 Contact / Bug Report

Feedback, questions or bug reports should be directed to Carl Leonardsson (`carl.leonardsson@it.uu.se`).

3 Installation

3.1 Requirements

1. A C++ compiler supporting C++11. For example g++ version 4.6 or higher.
2. In order to run the graphical interface, python is required at a version of 2.6 or higher installed with tcl/tk of version 8.4 or higher.
3. For predicate abstraction the MathSAT SMT solver as well as the library gmpxx are required. Memorax supports MathSAT 4 and MathSAT 5. MathSAT 4 is recommended. Memorax can be compiled without MathSAT and gmpxx, but will then not support predicate abstraction.
4. To be able to graphically draw automata, Graphviz is required.

3.2 Basic Installation

In the simplest case, Memorax can be installed with the following commands:

```
$ tar xvf memorax-<version>.tar.gz
$ cd memorax-<version>
$ ./configure
$ make
$ make install
```

3.3 Installation Options

The configure script is built with GNU autotools, and should accept the usual options and environment variables. This section outlines some of the typical use cases.

Changing Installation Directory The command 'make install' will install Memorax, its graphical interface and its documentation in the directories which are standard on your system. To override this behaviour add the switch --prefix to the './configure' command:

```
$ ./configure --prefix=/your/desired/install/path
```

Compiling with Predicate Abstraction Support To support predicate abstraction, Memorax must be compiled with MathSAT and gmpxx. Their header files and shared libraries must reside where they can be found by the compilation. If they are installed in non-standard locations, then the compilation can be directed to their location by appropriately specifying CXXFLAGS and LDFLAGS when invoking the './configure' command:

```
$ ./configure CXXFLAGS='-I/path/to/mathsat/include' \
              LDFLAGS='-L/path/to/mathsat/lib'
```

If MathSAT and/or gmpxx are not found by the configure script, then Memorax will be installed without support for predicate abstraction.

Specifying Compiler When the configure script is invoked, it will by GNU autotools magic determine which C++ compiler will be used during compilation. In case e.g. your default compiler does not support C++11, but you have the compiler g++-4.6 installed at a non-standard location you may want to override this. In order to do so, specify the path to g++-4.6 in CXX when invoking the './configure' command:

```
$ ./configure CXX='/path/to/g++-4.6'
```

3.4 Troubleshooting

MSatFailure In case you get the following error message when trying to use the PB abstraction:

```
Error: MSatFailure: Program is not compiled with MathSAT.
```

In order to use predicate abstraction, (e.g. the PB abstraction) Memorax needs to be compiled with MathSAT. To solve the problem, install MathSAT on your system and then reinstall Memorax. In case the installation fails to find MathSAT (see the output from the configure script), then try the instructions in the paragraph "Compiling with Predicate Abstraction Support" above.

4 Usage

Memorax provides a command line interface, as well as a graphical interface. It has three main modes of operation (henceforth "*commands*"): reachability checking, automatic fence inference and graphical representation of programs as automata. In all three modes Memorax works on parallel programs given in the Rmm language (See section 6.).

The algorithms used for reachability and automatic fence inference depend on which *abstraction* is selected. An abstraction defines what a configuration

looks like, what are the semantics of the analysed program and how the reachability analysis works. Abstractions can be over or under approximations of the semantics given in section 6.3. They can alternatively be exact, or even an approximation that is neither an over approximation nor an under approximation. Memorax currently supports two abstractions: SB and PB. See section 7 for details.

Reachability checking In this mode, Memorax will attempt to determine whether or not certain ("*forbidden*") configurations are reachable when the Rmm program is executed. The forbidden configurations are specified in the Rmm program as combinations of control states; one for each process. Any configuration where the processes are each in a control state such that they together satisfy such a combination is considered forbidden.

The reachability is determined by some reachability analysis, which depends on what abstraction is chosen.

Automatic fence inference In this mode, Memorax will perform repeated reachability checks, while gradually adding memory fences that turn out to be necessary in order to guarantee the non-reachability of the forbidden control states. Memorax will report a collection C of sets S of memory fences such that for every set S, the memory fences in S are sufficient to guarantee the non-reachability of the forbidden control states. Furthermore each fence in S is necessary, in the sense that adding all fences except one from S to the program, is insufficient to guarantee non-reachability. Here, "reachability" should be interpreted as reachability according to the given abstraction. Thus, an over approximating abstraction may report more fences than are actually necessary under the actual memory model, and an under approximating abstraction may fail to report fences that are actually necessary. The abstractions SB and PB (when run without a bound on the number of refinements) guarantee that the reported fences are both necessary and sufficient.

For the TSO memory model, reported fences are identified with write instructions. Adding the fence to the Rmm program corresponds to making that write instruction into a locked write instruction. In the actual machine code/assembly code that implements the program runnable on real hardware, this corresponds to adding memory fence immediately after the writing instruction. On x86, one can alternatively change the writing instruction into a LOCK'd version of the same instruction.

Graphical representation of Rmm programs In this mode, Memorax produces a PDF file containing a graphical representation of automata corresponding to the given Rmm program. There will be one automaton per process in the program.

4.1 Using the Command Line Interface

A call to the command line interface is on the following form:

```
memorax [command] [options] [program]
```

The `[command]` part indicates the mode of operation. It should be given as one of `reach` (indicating reachability analysis), `fencins` (indicating automatic fence inference) and `dotify` (indicating graphical representation of the RMM program).

The `[options]` part is optional and gives details about how the command should be executed. Accepted options are listed and explained below.

The `[program]` part should be the path to a text file containing an RMM program. The program path can be left out of the command line invocation, in which case MEMORAX will expect the RMM program via the standard input.

Options:

- `-o <filename>` or `--output <filename>`
  Write output to `<filename>`. This option is used to specify the desired path of the PDF file produced by the `dotify` command.

- `-a <abstraction>` or `--abstraction <abstraction>`
  Use abstraction `<abstraction>`. The abstraction should be one of `pb` and `sb`. If no abstraction is specified, then MEMORAX will default to using the SB abstraction.

- `-k <int>`
  Use `k` as buffer bound. The TSO buffers in the PB abstraction will not be allowed to grow larger than this many elements.

- `--cegar`
  Use CEGAR refinement in reachability analysis. CEGAR can be used with the PB abstraction, and will refine the abstraction by gradually, and as necessary, using additional predicates in the predicate abstraction, and a larger bound on the length of the TSO buffers.

- `--max-refinements <int>`
  Perform at most `<int>` many refinements in the CEGAR loop. If more refinements are necessary, then MEMORAX will terminate with an error message.

- `-v` or `--verbose`
  Print output verbosely.

- `-vv` or `--very-verbose`
  Print output very verbosely.

- `-vvv` or `--very-very-verbose`
  Print output very very verbosely.

- `-o1` or `--only-one`
  During fence insertion, stop searching after finding one sufficient, minimal fence set.

- `--rff`
  Convert machine to *register free form* before using it. Converting an RMM program to register free form, means to rewrite it such that the values of the registers are encoded in the control states, and all registers are replaced

by the corresponding integer literals wherever they occur in instructions. This conversion is possible when all registers in the program have finite domains. Converting a program to register free form may be beneficial for analysis time, in particular when using the SB abstraction.

4.2 Using the Graphical Interface

The graphical interface is a python script using tcl/tk, running on top of the command line interface. It runs on top of the command line interface, provides the CLI with appropriate switches and performs some interpretation of the output from the CLI. The graphical interface is installed as `memorax-gui`.

The GUI window contains, from top to bottom:

- A menu bar, allowing to load and save RMM programs and output, and to configure the behaviour of the GUI.
- A *command area* containing a number of buttons, check buttons etc.
- A *code area* where an RMM program may be loaded, edited and saved. All commands executed with the GUI will act on the program displayed here.
- An *output area* where text output from the underlying CLI will be displayed. The output is divided into two different consoles: "Output" and "Error".

At the top of the command area are radio buttons allowing the user to chose a command: "Reachability" (indicating reachability analysis), "Fence insertion" (indicating automatic fence inference) and "Draw automata" (indicating graphical representation of the RMM program).

For all commands, the user may specify a level of output verbosity ranging from "Only Results" (least output) to "Extreme" (most output). For most users and use cases, one of the levels "Only Results" and "Messages" is probably the most suitable.

Execution of the selected command is started with the button "Run". A running execution can be interrupted with the button "Break" (shortcut Ctrl-C). While the underlying tool is running, it will output text to the text fields "Output" and "Error" at the bottom of the GUI window.

For commands "Reachability" and "Fence Insertion", the user may chose what abstraction should be used: SB or PB. (See section 7.) Whether CEGAR should be used for automatically refining the PB abstraction. Also the user may chose to convert the RMM program into *register free form* before analysing it. Converting an RMM program to register free form, means to rewrite it such that the values of the registers are encoded in the control states, and all registers are replaced by the corresponding integer literals wherever they occur in instructions. This conversion is possible when all registers in the program have finite domains. Converting a program to register free form may be beneficial for analysis time, in particular when using the SB abstraction.

When the command "Fence Insertion" finds a set of fences, they are indicated in the output by textual representation of the writing transitions in the RMM automata which should be changed into locked writes. Mouse-over will highlight

the corresponding lines of code in the code area, and clicking them will center the code area over those lines.

For the command “Draw Automata”, the user should specify a path, where a PDF file displaying the RMM program will be created. When the “Draw Automata” command is used, the PDF file will be immediately displayed, provided that the user has specified a PDF viewer in the GUI configuration (Misc→Configuration).

5 Tutorial

This section gives a short tutorial to usage of MEMORAX.

Start the GUI.

```
$ memorax-gui &
```

The GUI window shows an example RMM program that can be analysed, an output area showing the version of the GUI, and a number of controls that allow the user to select a command and options.

Reachability analysis First, let us analyse the reachability of the forbidden states in the example program: Select the command “Reachability”, the abstraction “SB” and the verbosity “Messages”. Press the “Run” button to start the analysis.

In case the GUI is unable to find the MEMORAX CLI, then you will receive an error message:

```
Failed to start subprocess (...)
[Errno 2] No such file or directory
Failed to terminate subprocess.
Interrupted
```

If so, enter Misc→Configuration and setup the correct path to where you have installed the MEMORAX CLI.

If the GUI finds the CLI, you should instead receive a screenful of text describing the result. The most important part is the last section. It tells you that the forbidden states are reachable when the example program is executed under the TSO memory model. I.e. that the program is unsafe.

```
Reachability analysis results:
  Reachable:              Yes
  Generated constraints: 500
  Size of visited set:   216
  Time consumption:      0 s
```

You will also receive a “witness trace” showing *how* the forbidden states can be reached in the SB semantics.

Fence inference Now, let us see how MEMORAX can be used to automatically infer the fences that are necessary to make the example program safe. Select the command “Fence insertion”. Keep the abstraction “SB” and verbosity “Messages”. Press the “Run” button.

If all goes well you should receive an output like this:

```
$ /path/to/memorax fencins --json -v --abstraction sb
Currently examining fence set:
  (No fences)

Reachability analysis results:
  Reachable:             Yes
  Generated constraints: 500
  Size of visited set:   216
  Time consumption:      0.01 s

Cycles found in trace:
TsoCycle (complete):
  P0: update(var:0, P0)
  L14 P0: read: var:1 = 0
  L22 P1: locked{ write: var:1 := 1 }
  L23 P1: read: var:0 = 0

Currently examining fence set:
  L13 P0: write: x := 1
  L22 P1: write: y := 1

Reachability analysis results:
  Reachable:             No
  Generated constraints: 86
  Size of visited set:   39
  Time consumption:      0 s

Found 1 fence set:
Fence set #0:
  L13 P0: write: x := 1
  L22 P1: write: y := 1

Total time to insert fences: 0.01s.
```

Reading it from top to bottom, it tells us the following:

```
Currently examining fence set:
  (No fences)
```

The inference procedure starts without any inserted memory fences.

```
Reachability analysis results:
  Reachable:             Yes
  Generated constraints: 500
  Size of visited set:   216
  Time consumption:      0.01 s
```

Without any memory fences, the forbidden states are reachable. At the verbosity level “Messages”, the witness traces are omitted. If you want to see the traces, use e.g. “Debug” instead.

```
Cycles found in trace:
```

```
TsoCycle (complete):
  P0: update(var:0, P0)
  L14 P0: read: var:1 = 0
  L22 P1: locked{ write: var:1 := 1 }
  L23 P1: read: var:0 = 0

Currently examining fence set:
  L13 P0: write: x := 1
  L22 P1: write: y := 1
```

The inference procedure analyses the witness trace, and concludes that in order to prevent the example program from reaching the forbidden states by such an execution, two memory fences are necessary. The memory fences are "`L13 P0: write: x := 1`" and "`L22 P1: write: y := 1`". This notation should be interpreted as follows: `L13 P0: write: x := 1` is the writing instruction of process 0 that occurs at line 13 in the code. The corresponding fence, which is suggested by the inference procedure, should be placed immediately after this writing transition. In the Rmm language, inserting the fence is done by changing `write: x := 1` into `locked write: x := 1` in the code.

```
Reachability analysis results:
  Reachable:             No
  Generated constraints: 86
  Size of visited set:   39
  Time consumption:      0 s
```

The inference procedure attempts another reachability analysis, now with the two new fences inserted. This time it turns out that the forbidden states are not reachable, and the current fence set is sufficient for safety.

```
Found 1 fence set:
Fence set #0:
  L13 P0: write: x := 1
  L22 P1: write: y := 1
```

The inference procedure terminates, telling us that it detected exactly one minimal and sufficient set of memory fences:

{`L13 P0: write: x := 1`, `L22 P1: write: y := 1`}

Hovering the mouse over the fence set will highlight the corresponding write instructions in the code area. Clicking the fence set will center the code over the highlighted instructions.

Adding the fences Let us manually insert the fences, and then try again. Rewrite the code by adding "`locked`" in two places as shown below:

```
/* An example code */

forbidden
CS CS

data
x = 0 :  [0:1]
y = 0 :  [0:1]

process                          | process
text                             | text
L0:                              | L0:
[locked] write:  x := 1;         | [locked] write:  y := 1;
read:  y = 0;                    | read:  x = 0;
CS:                              | CS:
write:  x := 0;                  | write:  y := 0;
goto L0                          | goto L0
```

The previous fence inference result told us that this new version of the example code should be safe. To satisfy our curiosity and to see what it looks like when we run the fence inference procedure on an already safe program, let us try to run the fence insertion command again. We get the following result:

```
$ /path/to/memorax/build/bin/memorax fencins --json -v \
    --abstraction sb
Currently examining fence set:
  (No fences)

Reachability analysis results:
  Reachable:              No
  Generated constraints: 86
  Size of visited set:   39
  Time consumption:      0 s

Found 1 fence set:
Fence set #0:
  (No fences)

Total time to insert fences: 0s.
```

It tells us that the fence inference procedure starts with no fences (no fences except the ones that are explicitly part of the program). It runs the reachability analysis and finds that the forbidden states are not reachable; the program is safe. Memorax concludes by telling us that it found exactly one memory fence set that is necessary and sufficient: the empty set. I.e. as expected, the program is safe and requires no additional fences.

6 The Rmm language

The Rmm language allows to model a parallel program and specify safety properties that should hold.

The sometimes assembly-like syntax of the RMM language is motivated by the necessity, when analysing programs under relaxed memory models, of unambiguously specifying the order of memory accesses, and whether variables are stored in memory or in registers. Note that for conventional programming languages, such as e.g. C, the memory access ordering, register allocations, reuse of common sub-expressions and the like depend on the compiler (and compiler switches). Therefore, when trying to verify an implementation written in a high level language, it may be necessary to examine the machine code after compilation, or use inline assembly, to be certain that the model, written in RMM, and the compiled program, written in some high level language, correspond.

This section will start by introducing the RMM language by giving an example together with explanation. Then we continue by explaining the abstract machine on which an RMM program runs. We introduce control structures and informally explain about instructions under the Sequentially Consistent memory model. Finally we give formally and informally, the semantics of all instructions under the TSO memory model.

6.1 Machine Model and Memory Addressing

This section describes the abstract machine on which an RMM program is executed.

A *machine* $(\mathcal{P}, A, \mathcal{R}, \mathcal{X})$ consists of a memory with (shared) memory locations $\mathcal{X}$, and a set of processes $\mathcal{P}$ executing in parallel. Each process $p \in \mathcal{P}$ has a unique process identifier $pid(p) \in \{0, \cdots, |\mathcal{P}| - 1\}$. We will subsequently use p and $pid(p)$ interchangingly where there is no danger of confusion.

Automata Each process p is equipped with an automaton $A(p) = (\mathcal{Q}_p, \Delta_p)$ describing the program executed by p. The set $\mathcal{Q}_p = \{0, \cdots\}$ is the set of control states of the automaton. The set Δ_p is the set of transitions of the automaton. A transition $(q_0, instr, q_1)$ consists of a source control state q_0, a target control state q_1 and an RMM *instruction*. Instructions will be defined and given semantics in later sections.

Registers Each process p has a set of registers $\mathcal{R}(p)$. Registers $r \in \mathcal{R}(p)$ hold integer values and can only be accessed by the owning process p. Registers are not affected by memory model relaxations since they are not located in memory (and also are private). The differences between registers and memory locations are summarised in table 1.

Integer Domains Each memory location v and each register r has an associated domain $domain(v), domain(r) \subseteq \mathbb{Z}$. The domain is either the (infinite) set of integers $\mathbb{Z}$, or a finite interval $\{i, i+1, \cdots, j-1, j\}$.

Memory Addressing Memory locations in $\mathcal{X}$ are of two kinds: *local* and *global*. Both kinds are accessible for reading and writing by all processes, and there is no difference between the two kinds regarding memory model relaxation. The difference is purely in how they are addressed. A global memory location v has an alphanumerical name n, and is addressed by all processes by precisely that name. A local memory location v has an alphanumerical name n, but

	Memory locations	Registers
Integer values	Yes	Yes
Accessible by	All processes	Owning process
In arith. expr.	No	Yes
Write	`write`-instruction	Assignment instruction (E.g. `$r0 := $r1 + 1`) or assigning read (E.g. `read: $r0 := x`)
Read	`read`-instruction (assigning read or asserting read)	Use in arithmetic expression (E.g. `$r1 + 1`)
Names	Alphanumerical or alphanumerical followed by process specifier	`$` followed by alphanumerical

Table 1: Differences between memory locations and registers.

is also associated with one particular process $owner(v) \in \mathcal{P}$. When a process p accesses the local variable v, it should use the name n`[`$spec$`]` where $spec$ depends on $owner(v)$ and on p as described in table 2. For example: suppose that process 2 should access a local variable v by the name `x`. If $owner(v) = 1$ then the correct address of v for process 2 would be `x[1]`. If $owner(v) = 2$ then `x[my]` would be the address, and if $owner(v) = 5$ then `x[4]` would be the correct address.

condition	*spec*
$owner(v) < p$	$owner(v)$
$owner(v) = p$	`my`
$owner(v) > p$	$owner(v) - 1$

Table 2: Process p, when accessing variable v declared locally in process $owner(v)$, should use the name v`[`$spec$`]`.

Pointers As a third way of addressing memory, pointer expressions are allowed in Rmm. The syntax is `[`e`]`, where e is an arithmetic expression over literal integers and register values. A pointer `[`e`]` occurring in a statement $stmt$(`[`e`]`) in Rmm is really syntactic sugar for the following composed statement:

```
either{
  assume:  e = 0; stmt(v_0)
or
  ...
or
  assume:  e = n; stmt(v_n)
}
```

Here $v_0 \cdots v_n$ are all *global* memory locations in the program, in the order they were declared. Local memory locations cannot be accessed through

pointers.

6.2 An Example

```
 1: /* Dijkstra's lock */

 2: forbidden
 3:   CS CS

 4: data
 5:   turn = * : [0:1]

 6: process
 7: data
 8:   flag = 0 : [0:2]
 9: registers
10:   $flag = * : [0:2]
11:   $turn = * : [0:1]
12: text
13:   START:
14:   write: flag[my] := 1;
15:   read: $turn := turn;
16:   while $turn != 0 do{
17:     read: $flag := flag[0];
18:     if $flag = 0 then
19:       write: turn := 0;
20:     read: $turn := turn
21:   };
22:   write: flag[my] := 2;
23:   read: $flag := flag[0];
24:   if $flag = 2 then
25:     goto START;
26:   CS:
27:   write: flag[my] := 0;
28:   goto START
```

```
29: process
30: data
31:   flag = 0 : [0:2]
32: registers
33:   $flag = * : [0:2]
34:   $turn = * : [0:1]
35: text
36:   START:
37:   write: flag[my] := 1;
38:   read: $turn := turn;
39:   while $turn != 1 do{
40:     read: $flag := flag[0];
41:     if $flag = 0 then
42:       write: turn := 1;
43:     read: $turn := turn
44:   };
45:   write: flag[my] := 2;
46:   read: $flag := flag[0];
47:   if $flag = 2 then
48:     goto START;
49:   CS:
50:   write: flag[my] := 0;
51:   goto START
```

Figure 1: RMM model of two processes using Dijkstra's mutual exclusion protocol [5]

Figure 1 shows an RMM model of two processes using Dijkstra's mutual exclusion protocol. We will explain the format line by line.

The first line is a comment. Everything starting with `/*` continuing until `*/` is ignored by the parser.

Lines 2 and 3 declare the safety property. Every RMM file must start with such a declaration. The word **`forbidden`** is a reserved word. Line 3 tells us that the declared safety property states that at no time may simultaneously process 0 be at its control state labelled `CS` (line 26) and process 1 be at its control state labelled `CS` (line 49). The label names `CS` and `CS` are coincidentally the same, but refer to different processes and hence different control states. Additional lines like line 3 can be added provided that they are separated by semi-colons. Below we have added the safety properties that none of the processes may enter

its critical section (label CS) while the other process is at its initial control state (label START).

```
forbidden
  CS CS ;
  START CS ;
  CS START
```

Lines 4 and 5 declare a memory location called `turn`. The word `data` is a reserved word. After the word `data` comes a list of memory location declarations. The declaration `turn = * : [0:1]` starts with the name of the memory location. Then states (`= *`) that it may initially have any value in its domain. The last part (`: [0:1]`) specifies that the domain of the memory location is all integers from and including 0 up to and including 1. Below we have extended the declaration section to also declare a variable `x` with domain $\mathbb{Z}$ and initial value 0, and a variable `y` with the default domain (which is also $\mathbb{Z}$) and unspecified initial value.

```
data
  turn = * : [0:1]
  x = 0 : Z
  y = *
```

Lines 6 to 28 declare process 0. Lines 29 to 51 similarly declare process 1, and will not be separately explained. The word `process` on line 6 is a reserved word and informs us that a process declaration begins. The process declaration has three parts: data declaration (optional), register declaration (optional) and text declaration (mandatory).

Lines 7 and 8 is the data declaration for process 0. It declares a memory location named flag, with domain $\{0, 1, 2\}$ and initial value 0. This memory location is like the memory location `turn` that we declared earlier, in that it is accessible for both reading and writing to all processes and in that it is affected by the memory model. The only difference between memory locations declared at the top level (*global* memory locations) and memory locations declared inside a process declaration (*local* memory locations) is the naming. In order to access a global memory location, a process will use its name as it is. A local memory location *var* is accessed by its name and a specifier: *var*`[`*spec*`]`. The correct way of addressing local memory locations is described in section 6.1 and in particular in table 2.

Lines 9 to 11 declare the registers of process 0. Registers are similar to memory locations. They correspond to processor registers, so they are accessible only to the process owning them, and they are not affected by the memory model. In Rmm, registers have alphanumerical names preceded by a single `$` character.

The word `text` on line 12 informs that the program code begins.

The program code is a semi-colon separated sequence of statements. Each statement is optionally preceded by a process-unique label and a colon.

Line 13 declares a label `START` that identifies the control state immediately before execution of the first instruction.

Line 14, 19, 22 and 27 are memory writes. A value computed by arithmetic operations on literal integers and on values in registers is assigned to a memory

location. In this case literal integers 0, 1 and 2 are stored in the global memory location `turn` and the local memory location `flag` of process 0.

Line 15, 17, 20 and 23 are memory reads. The value in a memory location is loaded into a register. In the case of line 17, the value in the local variable `flag` of process 1 (see table 2) is loaded into the register `$flag` of process 0.

Lines 16 to 21 is a while-loop. The loop condition works on literal integers and values in registers.

Lines 18-19 and 24-25 are if-statements. The if-condition works on literal integers and values in registers. If-statements may optionally have an else-clause.

Lines 25 and 28 are goto-statements. A goto-statement `goto` *LBL* immediately redirects the control flow to the control state labelled by *LBL*.

6.3 Semantics

6.3.1 Statements

In the Rmm language a process's automaton is defined by a statement that follows the keyword `text` in the process declaration. Statements come in two kinds: *instructions* and *control structures*.

An *instruction* is a statement that can be attached to a transition in an automaton, and be executed atomically. An instruction corresponds loosely to a machine language instruction on an actual piece of hardware. But there are differences: Some composed operations that would require many machine language instructions can be performed in a single Rmm instruction. This is the case for local operations, such as evaluation of arithmetic expressions over private registers, where the non-atomicity of the operation on actual hardware is not observable. Furthermore, some instructions on hardware architectures, such as e.g. un-LOCK'd INC on Intel x86, appear as a single instruction in the machine language but executes equivalently to multiple subsequent and non-atomic memory accesses. Such instructions are not included in Rmm.

A *control structure* is a statement that affects the structure of the process automaton. Control structures themselves cannot occur as labels for individual transitions, but control structures may contain instructions and define a sub-automaton with transitions labelled by those instructions.

Table 3 lists all types of Rmm statements.

6.3.2 Arithmetical and Boolean Expressions

Some statements make use of arithmetical or boolean expressions. An arithmetical expression may contain registers, integer literals, addition, subtraction, unary minus and parentheses. A boolean expression may contain the boolean literals `true` and `false`, conjunction (`&&`), disjunction (`||`), negation (`not`), parentheses (`[]` is used for boolean expressions as opposed to `()` for arithmetical) and comparison of arithmetical expressions by the following comparison functions: `=`, `!=`, `<`, `>`. The expressions are interpreted in the obvious way. Note that memory locations cannot be used in expressions!

6.3.3 Instructions Informally

This section gives an informal description of the semantics of Rmm instructions under the SC memory model.

Instructions	
Name	**Example**
Nop	`nop`
Assignment	`$reg := 42`
Assume	`assume: $r0 = 0 && $r1 > 2`
Asserting read	`read: x = 3`
Assigning read	`read: $reg := x`
Write	`write: x := $r0 + $r1 - 1`

Locked block	`locked{`
	`  read: x = 0;`
	`  write: x := 1`
	`}`

(Locked write)	`locked write: x := 1`
(Compare & Swap)	`cas(x,2,13)`
Control statements	
Name	**Example**
Goto	`goto LBL`

Sequence	`{`
	`  read: $r0 := x;`
	`  $r1 := $r0 + 1;`
	`  write: x := $r1`
	`}`

If-statement	`if $r0 = 0 then {`
	`  $r1 := 1;`
	`  $r2 := 20`
	`} else`
	`  goto L0`

While-statement	`while $reg > 0 do` `  read: $reg := x`

Either-statement	`either{`
	`  read: v = 0`
	`or`
	`  read: v = 1;`
	`  write: w := 1`
	`}`

Table 3: RMM statements

In the below, we use the following conventions: Registers are named *reg*, *reg′* etc. Registers used in an instruction always refer to registers owned by the process that executes the instruction. Arithmetical expressions are named *expr*, *expr′* etc. Boolean expressions are named *bexpr*, *bexpr′* etc. Memory locations are named v, v' etc. That an instruction is *enabled* means that it can be executed. An instruction that is not enabled is blocking.

Nop `nop`

This instruction is always enabled, and has no effect when executed.

Assignment *reg* `:=` *expr*

Evaluates the expression *expr* and stores the result in the register *reg*. The instruction is enabled precisely when the valuation of *expr* is within the domain of *reg*.

Assume `assume:` *bexpr*

Is enabled precisely when *bexpr* evaluates to true. The instruction has no effect when executed.

Asserting Read `read:` v `=` *expr*

Is enabled when the value of memory location v in memory is the same as the value to which *expr* evaluates. The instruction has no effect when executed.

Assigning Read `read:` *reg* `:=` v

Reads the value of memory location v from memory and stores the value in register *reg*. The instruction is enabled precisely when the value of v in memory is within the domain of *reg*.

Write `write:` v `:=` *expr*

Evaluates the expression *expr* and writes the result to memory location v. Enabled when the value of *expr* is in the domain of v.

Locked Block

```
locked{
 sl_0
or
 ...
or
 sl_n
}
```

Here sl_i for all $0 \leq i \leq n$ is a semi-colon separated sequence of instructions. The sequence sl_i is said to be enabled if it is possible to execute its constituent instructions in order without blocking and without context-switching. When the locked block is executed, any one single enabled sequence sl_i is picked, and the instructions of sl_i are executed in order atomically. The locked block is enabled when there is at least one sequence sl_i that is enabled.

Important: The locked block is a powerful construction meant to enable modellers to model the occasional more obscure machine instructions that may occur on their hardware. Its TSO semantics are quite complicated, and improper

use is easy. It is therefore recommended to not explicitly use locked blocks unless absolutely necessary. It is recommended to limit use of locked blocks to implicit use by means of the two instructions *locked write* and *Compare & Swap*.

Locked Write `locked write:` v `:=` *expr*

Under SC semantics, a locked write is equivalent to an ordinary write. The locked write `locked write:` v `:=` *expr* is syntactic sugar for
`locked{ write:` v `:=` *expr*`}`

Compare & Swap (CAS) `cas(`v`,`*expr*`,`*expr*$'$`)`

The compare and swap instruction is enabled precisely when the value of v in memory is equal to the value of *expr*, and *expr*$'$ evaluates to a value within the domain of v. Executing the compare and swap instruction will store the value of *expr*$'$ in memory location v. The instruction `cas(`v`,`*expr*`,`*expr*$'$`)` is syntactic sugar for `locked{ read:` v `=` *expr*`; write:` v `:=` *expr*$'$`}`

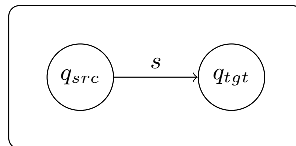
6.3.4 Control Structures

This section describes how control structures in the Rmm language are used to shape the process automata of a machine. This is done by describing the sub-automata corresponding to each type of control structure.

Below we use the following conventions: The described sub-automaton is a part of the automaton $A(p)$ corresponding to process p. The initial state of the sub-automaton is q_{src}. The control state that corresponds to the position in the Rmm code immediately after the control structure is named q_{tgt}.

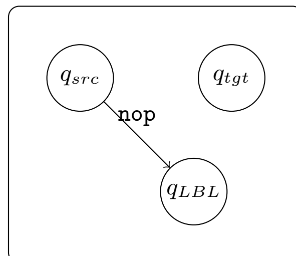
Instructions s

A single instruction s translates into a transition (q_{src}, s, q_{tgt}).



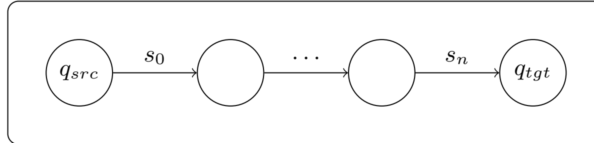
Goto `goto` LBL

Here LBL is a label attached to some control state q_{LBL} in $\mathcal{Q}_p$. The goto statement translates into a transition $(q_{src}, \texttt{nop}, q_{LBL})$.



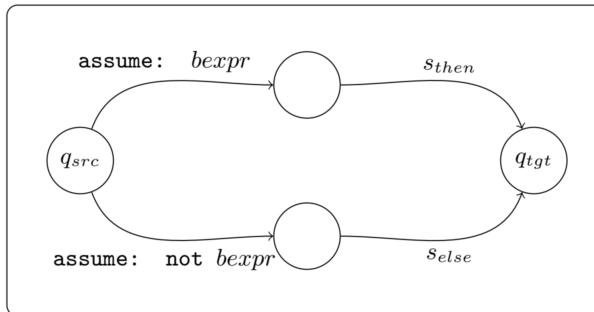
Sequence `{`s_0`;`$\cdots$`;`s_n`}`

The sequence construct arranges its constituent sub-statements, unsurprisingly, in a sequence from q_{src} to q_{tgt}, as shown in the diagram below.



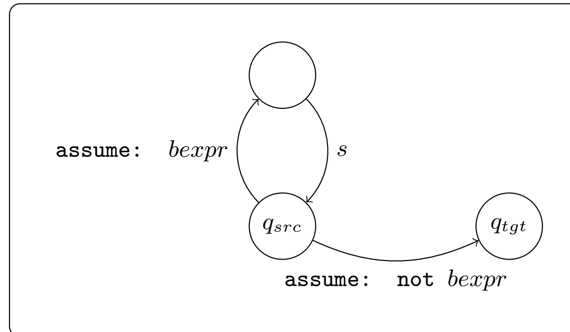
If-statement `if` *bexpr* `then` s_{then} `else` s_{else}

An if-statement branches the automaton into two branches where the initial assume transitions ensure that only one branch can be taken at any one time.



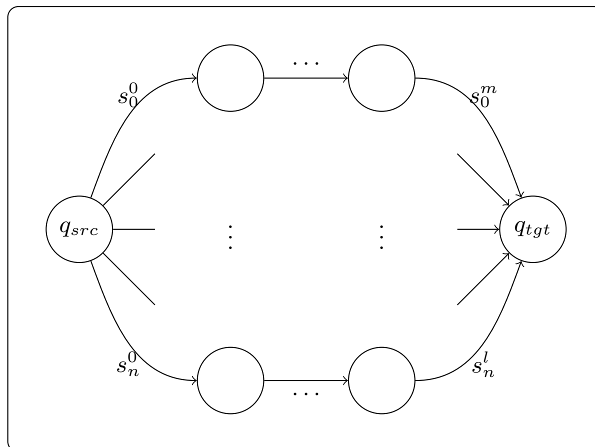
While-statement `while` *bexpr* `do` s

A while-statement translates to a loop that will be taken as long as *bexpr* evaluates to true, but no longer.



Either-statement `either{`s_0^0`;`$\cdots$`;`s_0^m `or` $\cdots$ `or` s_n^0`;`$\cdots$`;`s_n^l`}`

An either statement is a non-deterministic choice. It translates into multiple branches with no attached guards (except for what may occur in the constituents s_i^j).



6.3.5 TSO Semantics

This section describes the TSO semantics of the instructions in the RMM language. This is done formally and informally in parallel.

A *constraint* (M, M_{reg}, pc, B) describes the configuration of an abstract machine at a particular time. Each memory location $v \in \mathcal{X}$ has a particular value $M(v) \in domain(v)$. Each register r owned by each process p has a particular value $M_{reg}(p)(r) \in domain(r)$. Each process p is at a particular control state $pc(p) \in \mathcal{Q}_p$.

Furthermore, each process p is equipped with a FIFO buffer
$B(p) = \langle B(p)_0, \cdots, B(p)_n \rangle$ where $n = len(B(p)) - 1$. More recently inserted elements have a lower index. For all $0 \leq i \leq n$, the element $B(p)_i = (v, w) \in \mathcal{X} \times \mathbb{Z}$ is a pending write of process p to memory location v with value $w \in domain(v)$.

When a process executes a write to a memory location under TSO, it does not immediately update the memory, but instead it enqueues the write in its write buffer $B(p)$. Asynchronously, and without the active participation of the process p itself, the enqueued writes will be pushed, one by one in the same order they were enqueued, to memory and dequeued from the write buffer. The event of a write reaching memory is called an *update*.

In the below we describe the rules for process p to perform a transition t. For a function f, by $f[x := v]$ we denote the function f' such that $f'(y) = f(y)$ if $y \neq x$ and $f'(x) = v$. For a register valuation $M_{reg}(p)$ and an arithmetic expression *expr* over registers from $\mathcal{R}(()p)$, we let $M_{reg}(p)[expr]$ denote the evaluation of *expr* where each register r in *expr* evaluates to $M_{reg}(p)(r)$. We define similarly $M_{reg}(p)[bexpr]$ for a boolean expression *bexpr*.

Nop $t = (q_{src}, \texttt{nop}, q_{tgt})$

The nop instruction is the same under TSO semantics as under SC semantics: It does nothing.

$$\frac{pc(p) = q_{src}}{(M, M_{reg}, pc, B) \rightarrow_t (M, M_{reg}, pc[p := q_{tgt}], B)}$$

Assignment $t = (q_{src}, reg \texttt{ := } expr, q_{tgt})$

The assignment instruction is the same under TSO semantics as under SC semantics: It evaluates *expr* and assigns the result to the register *reg*.

$$\frac{\begin{array}{c} pc(p) = q_{src} \quad w \in domain(reg) \\ M'_{reg} = M_{reg}[p := M'_p] \\ \text{where} \\ w = M_{reg}(p)[expr] \\ M'_p = M_{reg}(p)[reg := w] \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M, M'_{reg}, pc[p := q_{tgt}], B)}$$

Assume $t = (q_{src}, \texttt{assume: } bexpr, q_{tgt})$

The assume instruction is the same under TSO semantics as under SC semantics: It evaluates *bexpr* and is enabled precisely when the result is true.

$$\frac{pc(p) = q_{src} \quad M_{reg}(p)[bexpr]}{(M, M_{reg}, pc, B) \rightarrow_t (M, M_{reg}, pc[p := q_{tgt}], B)}$$

Asserting Read $t = (q_{src}, \texttt{read: } v \texttt{ = } expr, q_{tgt})$

A read under TSO semantics will read the value of memory location v from memory, provided that the buffer of p does not contain any write to v. If there is a write to v in $B(p)$, then the value of the newest such write in $B(p)$ is read.

To formalise this, we define the function $read : ((\mathcal{X} \mapsto \mathbb{Z}) \times buffer) \mapsto \mathcal{X} \mapsto \mathbb{Z}$ as follows:

$$read(M, b)v =$$

$$\begin{cases} w & \text{If for some } i \in \mathbb{Z} \quad \left(\begin{array}{c} b_i = (v, w) \\ \text{and} \\ \neg\exists 0 \leq j < i, w' \in \mathbb{Z}.b_j = (v, w') \end{array} \right) \\ M(v) & \text{Otherwise} \end{cases}$$

Now we can define the transition rule:

$$\frac{pc(p) = q_{src} \quad read(M, B(p))v = M_{reg}(p)[expr]}{(M, M_{reg}, pc, B) \rightarrow_t (M, M_{reg}, pc[p := q_{tgt}], B)}$$

Assigning Read $t = (q_{src}, \texttt{read: } reg \texttt{ := } v, q_{tgt})$

An assigning read, reads the value of v from memory or from $B(p)$ in the same manner as an asserting read, but then assigns the read value to the register *reg*.

$$\frac{\begin{array}{c} pc(p) = q_{src} \quad w \in domain(reg) \\ M'_{reg} = M_{reg}[p := M'_p] \\ \text{where} \\ w = read(M, B(p))v \\ M'_p = M_{reg}(p)[reg := w] \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M, M'_{reg}, pc[p := q_{tgt}], B)}$$

Write $t = (q_{src}, \texttt{write: } v \texttt{ := } expr, q_{tgt})$

A write instruction evaluates the value w of the expression $expr$, and enqueues the write as (v, w) in its buffer.

$$\frac{\begin{array}{c} pc(p) = q_{src} \quad w \in domain(v) \\ B' = B[p := (v, w) \cdot B(p)] \\ \text{where} \\ w = M_{reg}(p)[expr] \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M, M_{reg}, pc[p := q_{tgt}], B')}$$

Update $t = update_p$

An *update* is not a transition in any process automaton. Instead it is an event that may happen at any time the buffer of process p is non-empty. When an update occurs, the oldest write (v, w) in the buffer of process p is dequeued, and the value of variable v is assigned w.

$$\frac{\begin{array}{c} B(p) = \langle B(p)_0, \cdots, B(p)_{n-1}, (v, w)\rangle \\ B' = B[p := \langle B(p)_0, \cdots, B(p)_{n-1}\rangle] \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M[v := w], M_{reg}, pc, B')}$$

Locked Write $t = (q_{src}, \texttt{locked write: } v \texttt{ := } expr, q_{tgt})$

A locked write acts as a write followed by a TSO fence. It requires the buffer of process p to be empty before it is executed. Then it evaluates the value w of $expr$ and writes w directly to v in memory without enqueueing the write in the buffer.

$$\frac{\begin{array}{c} pc(p) = q_{src} \quad B(p) = \langle\rangle \\ w \in domain(v) \\ \text{where} \\ w = M_{reg}(p)[expr] \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M[v := w], M_{reg}, pc[p := q_{tgt}], B)}$$

The locked write `locked write:` v `:=` $expr$ is syntactic sugar for `locked{ write:` v `:=` $expr$`}`.

CAS $t = (q_{src}, \texttt{cas(}v\texttt{,}expr\texttt{,}expr'\texttt{)}, q_{tgt})$

A compare and swap instruction acts as a fence in that it requires the buffer of process p to be empty before it can be executed. It then evaluates the values w and w' respectively for $expr$ and $expr'$, compares the value of v in memory with w, if the values are equal then the value w' is written to v in memory, otherwise the instruction blocks. A compare and swap instruction does not enqueue any write to the buffer.

$$\frac{\begin{array}{c} pc(p) = q_{src} \quad B(p) = \langle\rangle \\ M(v) = w \quad w' \in domain(v) \\ \text{where} \\ w = M_{reg}(p)[expr] \quad w' = M_{reg}(p)[expr'] \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M[v := w'], M_{reg}, pc[p := q_{tgt}], B)}$$

Locked Block $\texttt{locked\{}s_0^0\texttt{;}\cdots\texttt{;}s_0^{m_0}\ \texttt{or}\ \cdots\ \texttt{or}\ s_n^0\texttt{;}\cdots\texttt{;}s_n^{m_n}\texttt{\}}$

A locked block acts as a fence iff there is some s_i^j that is a write. If so, then the whole locked block is enabled only if the buffer of process p is empty.

When a locked block executes, it non-deterministically selects one sequence $s_i^0\texttt{;}\cdots\texttt{;}s_i^{m_i}$ and executing atomically all constituent instructions s_i^j in order. When doing so each constituent instruction executes as it normally does, except for writes, which execute as locked writes.

In order to formalise the above, we first define the functions *is_write* and *contains_write*:

$$is_write(s) = (\exists v \in \mathcal{X}, w \in \mathbb{Z}. s = \texttt{write:}\ v\ \texttt{:=}\ w)$$

$$contains_write(s) =$$

$$\left(\begin{array}{c} is_write(s) \\ \text{or} \\ \left(\begin{array}{l} \exists s_0^0, \cdots, s_0^m, \cdots, s_n^0, \cdots, s_n^l, i, j. \\ \quad s = \texttt{locked\{}s_0^0\texttt{;}\cdots\texttt{;}s_0^m\ \texttt{or}\ \cdots\ \texttt{or}\ s_n^0\texttt{;}\cdots\texttt{;}s_n^l\texttt{\}} \\ \quad \text{and} \\ \quad contains_write(s_i^j) \end{array}\right) \end{array}\right)$$

Next we define the locked transition relation $\rightarrow_s^{locked}$, for instructions s by the following two rules:

$$\frac{\begin{array}{c}\neg is_write(s) \\ (M, M_{reg}, pc, B) \rightarrow_{(pc(p),s,pc(p))} (M', M'_{reg}, pc', B')\end{array}}{(M, M_{reg}, pc, B) \rightarrow_s^{locked} (M', M'_{reg}, pc', B')}$$

$$\frac{\begin{array}{c} is_write(s) \\ (M, M_{reg}, pc, B) \rightarrow_{(pc(p),s,pc(p))} (M', M'_{reg}, pc', B') \\ (M', M'_{reg}, pc', B') \rightarrow_{update_p} (M'', M''_{reg}, pc'', B'')\end{array}}{(M, M_{reg}, pc, B) \rightarrow_s^{locked} (M'', M''_{reg}, pc'', B'')}$$

Now we are ready to define the transition rule for the locked block. Let $s = \texttt{locked\{}s_0^0\texttt{;}\cdots\texttt{;}s_0^{m_0}\ \texttt{or}\ \cdots\ \texttt{or}\ s_n^0\texttt{;}\cdots\texttt{;}s_n^{m_n}\texttt{\}}$.

$$\frac{\begin{array}{l} \quad pc(p) = q_{src} \\ (contains_write(s) \Rightarrow B(p) = \langle\rangle) \\ \exists 0 \le i \le n, c_0, \cdots, c_{m_i+1}. \\ \quad c_0 = (M, M_{reg}, pc, B) \\ \quad c_{m_i+1} = (M', M'_{reg}, pc, B') \\ \quad \forall 0 \le j \le m_i. c_j \rightarrow_{s_i^j}^{locked} c_{j+1} \end{array}}{(M, M_{reg}, pc, B) \rightarrow_t (M', M'_{reg}, pc[p := q_{tgt}], B')}$$

Important: The locked block is a powerful construction meant to enable modellers to model the occasional more obscure machine instructions that may occur on their hardware. Its TSO semantics are quite complicated, and improper use is easy. It is therefore recommended to not explicitly use locked block unless absolutely necessary. It is recommended to limit use of locked blocks to implicit use by means of the two instructions *locked write* and *Compare & Swap*.

6.4 Grammar

```
RMM ::=
    BAD-STATES PREDICATES 'data' VAR-INIT PROCESS-LIST
  | BAD-STATES PREDICATES PROCESS-LIST

BAD-STATES ::= 'forbidden' BAD-STATES-LIST-LIST

BAD-STATES-LIST-LIST ::=
    BAD-STATES-LIST
  | BAD-STATES-LIST ';' BAD-STATES-LIST-LIST

BAD-STATES-LIST ::=
    LABEL
  | LABEL BAD-STATES-LIST

PREDICATES ::=
    ''
  | 'predicates' BEXPR-LIST

BEXPR-LIST ::=
    BEXPR
  | BEXPR ';' BEXPR-LIST

PROCESS-LIST ::=
    PROCESS
  | PROCESS PROCESS-LIST

PROCESS ::=
  'process' PROC-COUNT VAR-INIT REG-INIT 'text' STMT-LIST

PROC-COUNT ::=
    ''
  | '(' NAT ')'

VAR-INIT ::=
    ''
  | 'data' VAR-INIT-LIST

VAR-INIT-LIST ::=
    ID '=' VAR-INIT-VALUE VAR-DOMAIN
  | ID '=' VAR-INIT-VALUE VAR-DOMAIN ',' VAR-INIT-LIST

VAR-INIT-VALUE ::=
    NAT
  | '-' NAT
  | '*'

VAR-DOMAIN ::=
    ''
```

```
  | ':' '[' INT ':' INT ']'
  | ':' 'Z'

REG-INIT ::=
    ''
  | 'registers' REG-INIT-LIST

REG-INIT-LIST ::=
    REG '=' VAR-INIT-VALUE VAR-DOMAIN
  | REG '=' VAR-INIT-VALUE VAR-DOMAIN ',' REG-INIT-LIST

STMT-LIST ::=
    LSTMT
  | LSTMT ';' STMT-LIST

LSTMT ::=
    STMT
  | LABEL ':' STMT

STMT ::=
    'nop'
  | 'read:' MEMLOC '=' EXPR
  | 'read:' REG ':=' MEMLOC
  | 'write:' MEMLOC ':=' EXPR
  | 'locked write:' MEMLOC ':=' EXPR
  | 'cas(' MEMLOC ',' EXPR ',' EXPR ')'
  | REG ':=' EXPR
  | 'assume:' BEXPR
  | 'if' BEXPR 'then' LSTMT
  | 'if' BEXPR 'then' LSTMT 'else' LSTMT
  | 'while' BEXPR 'do' LSTMT
  | 'goto' LABEL
  | 'either' '{' STMT-LIST EITHER-LIST '}'
  | 'locked' '{' STMT-LIST EITHER-LIST '}'
  | '{' STMT-LIST '}'

EITHER-LIST ::=
    ''
  | 'or' STMT-LIST EITHER-LIST

BEXPR ::=
    BEXPR-AND
  | BEXPR '||' BEXPR

BEXPR-AND ::=
    BEXPR-ATOM
  | BEXPR-AND '&&' BEXPR-AND
  | 'not' BEXPR-ATOM

BEXPR-ATOM ::=
```

```
    'true'
  | 'false'
  | EXPR '=' EXPR
  | EXPR '!=' EXPR
  | EXPR '<' EXPR
  | EXPR '>' EXPR
  | '[' BEXPR ']'

EXPR ::=
    EXPR '+' EXPR
  | EXPR '-' EXPR-UNIT
  | EXPR-UNIT

EXPR-UNIT ::=
    REG
  | NAT
  | '-' EXPR-UNIT
  | '(' EXPR ')'

REG ::= $[_a-zA-Z0-9]+

MEMLOC ::=
    ID
  | ID '[' 'my' ']'
  | ID '[' NAT ']'
  | '[' EXPR ']'

LABEL ::= ID

ID ::= [_a-zA-Z][_a-zA-Z0-9]*
```

6.5 Macros

The parsing of Rmm code involves a preprocessing step, where macros can be defined and called. This allows for example, to define a process as a macro, then instantiate it multiple times with different arguments, to avoid typing similar processes definitions multiple times.

The syntax of macro definition is as follows, where *mname* is some identifier which is the name of the defined macro, $p_0, \cdots, p_n$ are identifiers which are the formal parameters of the macro, and *mbody* is some Rmm code that may contain $p_0, \cdots, p_n$ at any point as a replacement for some sequence of symbols. The body *mbody* may not contain the keyword `endmacro`.

`macro` $mname(p_0, \cdots, p_n)$
$mbody(p_0, \cdots, p_n)$
`endmacro`

```
forbidden
 CS CS

data
 cs0 = 0 :  [0:1]
 cs1 = 0 :  [0:1]
 x = 0   :  [0:1]
 y = 0   :  [0:1]

macro p(x,y,pid)
process
text
 write:  x := 1;
 read:  y = 0;
 CS: write:  [pid] := 1
endmacro

p(x,y,0)
p(y,x,1)
```

```
forbidden
 CS CS

data
 cs0 = 0 :  [0:1]
 cs1 = 0 :  [0:1]
 x = 0   :  [0:1]
 y = 0   :  [0:1]

process
text
 write:  x := 1;
 read:  y = 0;
 CS: write:  cs0 := 1

process
text
 write:  y := 1;
 read:  x = 0;
 CS: write:  cs1 := 1
```

Figure 2: Left: RMM code using macros. Right: Equivalent, expanded code.

A macro call may occur at any position in an RMM code, after the called macro has been completely defined. A macro call has the following syntax, where *mname* is the name of some defined macro of arity $n+1$, and each a_i is some sequence of symbols.

mname`(`a_0`,`$\cdots$`,`a_n`)`

Each sequence of symbols a_i must be well-formed with respect to `(` and `)`, and may not contain a comma (`,`) except if it is within some nesting of `(` and `)`. Cyclic macro calls are not allowed.

Figure 2 shows an example of macro usage.

7 Abstractions

7.1 PB - Bounded Buffers with Predicate Abstraction

The PB abstraction is the implementation of [2]. It is an over approximation of TSO.

The PB abstraction uses an over approximation of the TSO store buffers. For a positive integer k, it stores the k most recent messages for each memory location and process. The information of older messages is dropped.

Predicate abstraction is used to enable (infinite) integer domains for memory locations and registers.

The reachability analysis is by backward state space exploration.

If CEGAR is used, then the value of k as well as the set of predicates for predicate abstraction is gradually refined. When CEGAR is used, analysis and fence insertion with PB is sound, but not complete. For fence insertion this means that any fence sets reported by MEMORAX are sufficient and minimal for preventing reachability of the forbidden states.

For details about the PB abstraction, see [2].

7.2 SB - Single Buffer

The SB abstraction is a reimplementation of [3]. Reachability analysis and fence insertion with SB is both sound and complete.

The SB abstraction defines a program semantic which is equivalent to TSO with regards to control state reachability. For the SB semantics, the well-quasi ordering framework [1] is applicable and provides a sound and complete reachability analysis.

The SB semantics replaces the TSO store buffers with a single, shared store buffer. Each message in the single store buffer contains a complete memory snapshot.

The reachability analysis is by backward state space exploration.

For details about the SB abstraction, see [3].

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, 1996.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, volume 7460 of *LNCS*. Springer, 2012.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under tso. In *TACAS*, 2012.

[4] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996.

[5] N. Lynch and B. Patt-Shamir. DISTRIBUTED ALGORITHMS , Lecture Notes for 6.852 FALL 1992. Technical report, MIT, Cambridge, MA, USA, 1993.

GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed

under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML

using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in

an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation

of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.