MEMORAX, a precise and sound tool for automatic fence insertion under TSO *

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Carl Leonardsson[1], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
[2] Academia Sinica, Taiwan
[3] Linköping University, Sweden

Abstract. We introduce MEMORAX, a tool for the verification of control state reachability (i.e., safety properties) of concurrent programs manipulating finite range and integer variables and running on top of weak memory models. The verification task is non-trivial as it involves exploring state spaces of arbitrary or even infinite sizes. Even for programs that only manipulate finite range variables, the sizes of the store buffers could grow unboundedly, and hence the state spaces that need to be explored could be of infinite size. In addition, MEMORAX incorporates an interpolation based CEGAR loop to make possible the verification of control state reachability for concurrent programs involving integer variables. The reachability procedure is used to automatically compute possible memory fence placements that guarantee the unreachability of bad control states under TSO. In fact, for programs only involving finite range variables and running on TSO, the fence insertion functionality is complete, i.e., it will find all minimal sets of memory fence placements (minimal in the sense that removing any fence would result in the reachability of the bad control states). This makes MEMORAX the first freely available, open source, push-button verification and fence insertion tool for programs running under TSO with integer variables.

1 Introduction

We introduce MEMORAX, the first freely available, open source (`https://github.com/memorax/memorax`), push-button verification and fence insertion tool that can handle integer variables and that is both sound and complete under TSO for all programs that only involve finite range variables. Modern concurrent processor architectures allow *weak* (*relaxed*) memory models, in which certain memory operations may overtake each other. The use of weak memory models makes reasoning about behaviours of concurrent programs challenging, even for skilled developers. This is for instance witnessed by the lively debate among developers on the Linux Kernel Mailing list about the correctness on x86 of the "Linux Ticket Lock" protocol. (See the mail thread starting with `https://lkml.org/lkml/1999/11/20/76`.) In fact, several synchronisation algorithms, such as mutual exclusion and producer-consumer protocols, turn out to be

* This research was in part funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC).

incorrect if run without modification on weak memories [5]. MEMORAX is based on the techniques developed in [4] and extended in [3]. Not only does our tool turn this verification task into a push-button exercise, it also automatically inserts fences in order to ensure correctness of programs that were made incorrect by the weak memory relaxation. More precisely:

- MEMORAX is an open source [2] push-button tool that comes with a graphical user interface and a simple low level language with a well defined semantics.
- it is sound for concurrent programs running on the TSO memory model (i.e., x86 and SPARC platforms) and involving variables with finite or integer ranges.
- it performs reachability on infinite state spaces to verify control state reachability.
- it provides users with concrete counter-examples, useful for debugging, that take the program from an initial configuration to a specified bad control state.
- it is complete, using an intricate encoding based on the theory of well-quasi-ordering, for the reachability problem of programs on TSO, provided that they only have finite range variables.
- it uses an off-the-shelf SMT solver (MathSAT [1]) to incorporate an interpolation based CEGAR loop to handle integer variables.
- it automatically finds (sets of) fences to ensure a safety property is respected if the property does hold on SC.
- it finds all minimal sets of fences for programs with finite range variables and running on TSO.

Targeted user base We see three potential groups of users for MEMORAX :

1. Computer science researchers can use the open source code of MEMORAX to compare with other approaches for the verification of programs running on top of weak memory models, to improve and optimise the implemented techniques (e.g. by interfacing with other SMT solvers or by improving the used data structures or the symbolic representations), or to target new platforms and programs (e.g. add soundness for RMO or PSO, or scale for heap manipulating programs)
2. Teachers of architecture and concurrent programming classes can use (and augment) MEMORAX with its simple user interface in order to familiarise their students with weak memory models. In particular the precision and counter example capabilities of MEMORAX can concretely illustrate the effects of relaxed memory.
3. Software developers working on complex and low level, lock-free code can use MEMORAX to easily check the effects of TSO on their tentative solutions. The generated error traces are also possible on weaker memory models and can conveniently help to highlight possible problems.

Related tools and approaches As far as we know, MEMORAX is the first available open source verification and fence insertion tool that is sound on TSO, that can handle integer variables, and that is complete for programs with finite range variables under TSO. There exists several very conservative approaches that restrict to SC executions by establishing “triangular race freedom” [12] or by inserting fences using “delay set

analysis" [13]. We will not further elaborate on those techniques, but will instead focus on a number of tools and approaches more similar to our own.

CheckFence [6] is a SAT-based tool that tests correctness of fence placements by considering finite executions on different relaxed memory models. The tool cannot verify programs that result in buffers of arbitrary size like the ones MEMORAX handles since it unrolls loops and checks correctness of the resulting finite executions.

Fender [8, 9] combines model checking with abstraction in order to perform reachability analysis on finite over-approximations. It considers different memory models and uses the reachability analysis to justify fence placements. The analysis is not exact and cannot guarantee to show absence of errors for correct programs. As a result, the tool lacks the precision that would allow it to find minimal sets of fence placements. Unfortunately, we were not able to find the tool which is, as far as we know, not open source. Finally, the tool does not handle, programs with integer variables.

mmchecker [7] performs explicit model-checking for the .NET memory model. It explores the (possibly infinite) state space and inserts fences in order to forbid behaviours that are not possible under SC. The tool cannot prove correctness of programs that generate infinite state spaces but do not require fences. Also the tool cannot soundly handle integer variables like MEMORAX does on TSO.

Automata based accelerations [10, 11] computes under-approximations of the generated infinite state space on different relaxed memory models. When the analysis terminates, it answers exactly whether the property is violated or not, and it allows to deduce minimal sets of fence placements, even for programs that may generate buffers of arbitrary sizes. The approach targets systems that manipulate finite variables. It neither can handle integer variables nor does it guarantee termination. We were not able to get hold of the tool or of its source code.

```
1  forbidden
2    CS CS
3
4  data
5    turn = * : [0:1]
6    x    = 0 : [0:1]
7    y    = 0 : [0:1]
```

```
9  process
10 registers
11   $r0 = * : [0:1]
12   $r1 = * : [0:1]
13 text
14   L0: write: x := 1;
15    write: turn := 1;
16   L1: read: $r0 := y;
17    read: $r1 := turn;
18    if $r0 = 1 && $r1 = 1 then
19      goto L1;
20   CS: write: x := 0;
21    goto L0
```

```
23 process
24 registers
25   $r0 = * : [0:1]
26   $r1 = * : [0:1]
27 text
28   L0: write: y := 1;
29    write: turn := 0;
30   L1: read: $r0 := x;
31    read: $r1 := turn;
32    if $r0 = 1 && $r1 = 0 then
33      goto L1;
34   CS: write: y := 0;
35    goto L0
```

Fig. 1. Peterson's mutual exclusion protocol.

2 Using the tool

2.1 The RMM language

Programs to be tested with MEMORAX are written in the special purpose language RMM. For reasoning about programs under relaxed memory, detailed knowledge about

how variables are stored and used is necessary. RMM is designed to unambiguously describe that aspect.

As an example, Figure 1 shows an RMM model of the Peterson mutual exclusion protocol. Lines 1-2 are of particular interest, since they specify the safety criterion: It is forbidden for the processes (henceforth called `P0` and `P1`) to simultaneously be in the control states labelled `CS` (i.e. line 20 for `P0` and line 34 for `P1`).

2.2 Usage through the Graphical Interface

The GUI is a python script (`memorax-gui`) wrapping around the CLI. The GUI window consists of three main parts: The command input area, the code area and the output area. The command input area allows to chose between the commands "Reachability", "Fence insertion" and "Draw automata", and specify options for the commands. All commands apply to the code in the code area, and print their output (and possibly errors) to the output area.

A typical work flow would be the following: First write the RMM code for the protocol you want to analyse. Then use the "Draw automata" command to produce a PDF file showing the automata for the defined processes. This is useful for asserting that the RMM code specifies what you intended. Next use the "Reachability" command to check whether the protocol is safe from the start. If not, then use the "Fence insertion" command to receive sets of fences that will make the protocol safe.

Reachability The Reachability command is used to analyse whether there is some configuration which violates the safety specification, but is reachable from some initial configuration. If there is such a configuration, then an error trace will be supplied.

There are currently two reachability methods ("abstractions") available in MEMORAX: SB and PB, corresponding respectively to our works in [4] and [3]. The PB method is an over-approximation and allows for CEGAR abstraction refinement.

Protocols can be rewritten to "*Register Free Form*" before being analysed. This encodes register values in control states, and can often improve analysis performance.

Fence insertion The fence insertion command will repeatedly execute reachability queries, while gradually adding fences to the analysed protocol in order to guarantee satisfaction of the safety criterion. The available options for fence insertion are the same as for reachability, and apply to the repeated reachability queries.

Interpreting the Output: If we apply the fence insertion command to the program in Figure 1, we will get output describing the results of the reachability queries. There will be a description of the result at the end of the output:

```
Found 1 fence set:
Fence set #0:
  L15 P0: write: turn := 1
  L29 P1: write: turn := 0
```

Here MEMORAX has found exactly one minimal and sufficient set of fences, namely the one corresponding to locking the writes at line 15 and 29. Other possible outcomes include the empty set - meaning the program is already correct, and no sets - meaning the program cannot be corrected with fences.

3 Implementation

MEMORAX is implemented in C++ with the intent of being easy to extend with new memory models and analysis methods.

Reachability optimisations. We mention some of the techniques we use to combat the state space explosion problem:

- *Light-Weight Pre-Analysis.* Before the reachability analysis is started, we apply a light-weight, per-thread, over-approximating analysis. This allows us to collect a rough invariant about the buffer contents that are possible per control state and process. We use the invariant to efficiently reduce the explored state space.
- *Update Restriction.* We soundly limit store buffer updating to only take place after a read instruction by the same process. The rationale is that it is only relevant to delay a write instruction by buffering if it is delayed past a read instruction. Other delays can be simulated under SC.
- *Partial Order Reduction for TSO.* In addition to the above update limitation, MEMORAX uses a partial order reduction technique based on the principle that an instruction reordering that does not participate in a conflict cycle, as defined in [13], can be simulated by an appropriate scheduling under SC. Thus instruction reorderings that do not participate in conflict cycles need not be analysed.

Fence insertion. The fence insertion algorithm relies on the underlying reachability analysis when evaluating each fence set placement. It is therefore desirable to keep the number of tried fence sets as small as possible.

- *Fence Placement Restriction.* We restrict the number of possibilities, by only considering fences that can be added by locking some write instruction. For example, changing `write: x := 1` into `locked write: x := 1` adds a fence after `write: x := 1`. This guarantees finding minimal and sufficient fence sets (if they exist). Their size can however be larger than a smallest sufficient set.
- *Multiple Fence Extraction.* We perform an extensive analysis to capture fences that need to be added in order to avoid a given error trace. By identifying the conflict cycles (as described by [13]) that a particular reordering $(a \to b)$ participates in, it is sometimes possible to deduce the existence of another, similar error trace where a and b occur in program order, but another pair $(c \to d)$ is reordered, yielding the same conflict cycle. In such cases a fence between c and d is equally necessary as a fence between a and b. Thus the fence insertion algorithm can infer more than one fence at a time, and the number of reachability queries can be decreased.

4 Experimental Results

Table 1 displays the results of running MEMORAX on several classical examples. For each of the examples, we give the total time for finding all minimal, sufficient sets of fences, using the methods SB and PB, with and without transforming the program

to register free form. In the table, "not-applicable" denotes that the corresponding approach is not applicable to the example. These correspond to applying a finite domain technique (SB and RFF) to an infinite domain program. Furthermore, out-of-mem is used to denote that the experiment failed to finish before consuming all available memory of the host computer. All examples were run on a laptop with a 2.27 GHz processor and 4 GB of memory.

	Size Proc./States/ Var./Trans.	Total time seconds				Fences necessary (smallest set)
		SB	SB(rff)	PB	PB(rff)	
1. Simple Dekker	2/6/2/6	0.0	0.0	0.0	0.0	1 per proc
2. Full Dekker	2/22/3/28	0.4	0.2	0.1	0.1	1 per proc
3. Peterson	2/12/3/14	1.9	1.0	3.5	0.4	1 per proc
4. Lamport Bakery (bounded)	2/18/4/20	out-of-mem	61.2	152.7	17.9	2 per proc
5. Lamport Fast	2/24/4/34	233.7	223.4	2.7	2.5	2 per proc
6. CLH Queue Lock	2/30/4/42	out-of-mem	15.4	out-of-mem	out-of-mem	0
7. Sense Reversing Barrier	2/4/2/4	0.3	0.2	0.1	0.0	0
8. Burns	2/8/2/9	0.0	0.0	0.0	0.0	1 per proc
9. Dijkstra	2/22/3/28	out-of-mem	0.4	1.0	2.0	1 per proc
10. Lamport Bakery (unbounded)	2/18/4/20	not-applicable	not-applicable	166.2	not-applicable	2 per proc
11. Linux Ticket Lock (unbounded)	2/4/2/4	not-applicable	not-applicable	0.4	not-applicable	0

Table 1. Experimental Results

References

1. *MathSAT4*. http://mathsat4.disi.unitn.it/.
2. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. `https://github.com/memorax/memorax`.
3. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, pages 164–180, 2012.
4. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Counter-example guided fence insertion under tso. In *TACAS*, 2012.
5. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996.
6. S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.
7. T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *Formal Methods (FM)*, LNCS 4085. Springer, 2006.
8. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FM-CAD*, 2011.
9. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.
10. A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN*, 2010.
11. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, 2011.
12. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*. 2010.
13. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. In *Transactions on Programming Languages and Systems*, volume 10, pages 282–312. ACM, 1988.

A Installation

This appendix is intended to guide the reader to use of MEMORAX. It will first describe installation of MEMORAX, then the modelling language RMM and finally give a tutorial using the graphical interface of MEMORAX. More details are available in the manual (`memorax/doc/manual.pdf`).

MEMORAX can be downloaded from `https://github.com/memorax/memorax`. The sources including documentation are available as a tar ball via the "Downloads" section (recommended) as well as by git clone.

A.1 Requirements

- A C++ compiler supporting C++11. For example g++ version 4.6 or higher.
- 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.
- 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.
- To be able to graphically draw automata, Graphviz is required.

On Debian based Linux systems, most of the requirements can be met by installing packages `g++` (for some version $\geq$ 4.6), `libgmp-dev`, `graphviz` and `python-tk`. The MathSAT SMT solver has to be separately downloaded and installed. The recommended version is 4 (`http://mathsat4.disi.unitn.it/`), but installation will also work with version 5 (`http://mathsat.fbk.eu/`).

A.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
```

When invoking the `configure` script, one should ensure that the installation scripts find the shared library and the header file for MathSAT. In case MathSAT is installed in a non-standard location the paths to `mathsat/include` and `mathsat/lib` must be passed to `configure` as shown in Figure 2.

B RMM Example

This section will introduce the RMM language by giving an example, and explaining it line by line. Figure 3 shows an RMM model of two processes using Dijkstra's mutual exclusion protocol.

```
$ sudo apt-get install g++ libgmp-dev graphviz python-tk
$ mkdir memorax ; cd memorax ; mkdir usr
$ wget http://mathsat4.disi.unitn.it/download.php\
      ?file=mathsat-4.2.17-linux-x86.tar.gz
$ tar xvf mathsat-4.2.17-linux-x86.tar.gz
$ wget https://github.com/downloads/memorax/memorax/memorax-0.1.1.tar.gz
$ tar xvf memorax-0.1.1.tar.gz
$ cd memorax-0.1.1
$ ./configure --prefix /path/to/memorax/usr \
      CXXFLAGS=-I/path/to/memorax/mathsat-4.2.17-linux-x86/include \
      LDFLAGS=-L/path/to/memorax/mathsat-4.2.17-linux-x86/lib
$ make
$ make install
```

Fig. 2. Local installation example on 32 bit Ubuntu 12.04. For the 64 bit version, just substitute `linux-x86` by `linux-x86_64` in the above.

Lines 1 and 2 declare the safety property. Every RMM file must start with such a declaration. The word `forbidden` is a reserved word. Line 2 tells us that the declared safety property states that at no time may simultaneously process 0 be at its control state labelled `CS` (line 27) and process 1 be at its control state labelled `CS` (line 51). The label names `CS` and `CS` are coincidentally the same, but refer to different processes and hence different control states. Additional lines like line 2 can be added provided that they are separated by semi-colons.

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 take on any value from its domain as its initial value. 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.

Lines 7 to 29 declare process 0. Lines 31 to 53 similarly declare process 1, and will not be separately explained. The word `process` on line 7 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 8 and 9 constitute 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 to all processes for both reading and writing, 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*]. A process p_0 that accesses a memory location belonging to a process p_1 will use $spec = \texttt{my}$ if $p_0 = p_1$, it will use $spec = p_1$ if $p_1 < p_0$ and finally $spec = (p_1 - 1)$

```
1 forbidden
2   CS CS
3
4 data
5   turn = * : [0:1]
```

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

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

Fig. 3. RMM model of two processes using Dijkstra's mutual exclusion protocol

if $p_1 > p_0$. This means that in the example both processes use `flag[0]` to access the memory location `flag` belonging to the other process.

Lines 10 to 12 declare the registers of process 0. Registers in RMM 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 13 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 14 declares a label `START` that identifies the control state immediately before execution of the first instruction.

Line 15, 20, 23 and 28 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 16, 18, 21 and 24 are memory reads. The value in a memory location is loaded into a register. In the case of line 18, the value in the local variable `flag` of process 1 is loaded into the register `$flag` of process 0.

This illustrates one of the two kinds of read instructions that are available in RMM. The other kind is a read of the form (`read:` *var* `=` *expr*), which blocks execution unless the value read from the memory location *var* equals the value of the arithmetic expression *expr* over literal integers and register values. The blocking reads can be used as modelling constructs either to force a particular read, or to simulate a spin loop that waits for a variable to take on a particular value. Use of blocking reads is shown in Figure 5.

Lines 17 to 22 is a while-loop. The loop condition works on literal integers and values in registers.

Lines 19-20 and 25-26 are if-statements. The if-condition works on literal integers and values in registers. If-statements may optionally have an else-clause.

Lines 26 and 29 are goto-statements. A goto-statement `goto` *LBL* immediately redirects the control flow to the control state labelled by *LBL*.

C MEMORAX Tutorial

This section gives a short tutorial to usage of MEMORAX, through its graphical interface. Throughout the tutorial we are going to use the small example program that is shown by the GUI at startup. This example contains no registers and only finite domain memory locations. For more detailed information about infinite domain analysis (using PB) and register free form, we refer to the manual (`memorax/doc/manual.pdf`) and our previous work [3].
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.

Draw automata Before starting to analyse an RMM program, we can use the "Draw automata" command to visualise the program as state machines. This is useful to ensure that the RMM code accurately models the intended protocol. It can also conveniently display the effect of rewriting a program to Register Free Form.

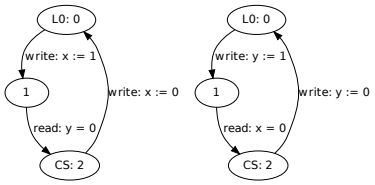

Fig. 4. State machines for the example RMM code.

Select the command "Draw automata". In the "Output" field, type a name for the PDF file that will be produced. Then press the "Run" button. The state machines corresponding to the example RMM code is now written to the indicated PDF file. Figure 4 shows the result. To automatically open PDF files when they are produced, you can set your preferred PDF viewer in Misc→Configuration.

Reachability analysis Let us now 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 our small example 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.

You will now receive a long output detailing the process of fence insertion, and the result. We will explain the output from top to bottom:

```
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:

$$\{\texttt{L13 P0: write: x := 1}, \texttt{L22 P1: write: y := 1}\}$$

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. Rewrite the code by adding "`locked`" in two places as shown in Figure 5. If we were to apply fence insertion again, to the corrected program, MEMORAX would tell us that the empty fence set is now sufficient; No additional fences are necessary.

```
1 forbidden
2   CS CS
3
4 data
5   x = 0 : [0:1]
6   y = 0 : [0:1]
8 process
9 text
10 L0:
11   locked write: x := 1;
12   read: y = 0;
13 CS:
14   write: x := 0;
15   goto L0
17 process
18 text
19 L0:
20   locked write: y := 1;
21   read: x = 0;
22 CS:
23   write: y := 0;
24   goto L0
```

Fig. 5. Correctly fenced example code.