

A CHR-Based Solver for Weak Memory Behaviors*

Allan Blanchard^{1,2}, Nikolai Kosmatov¹, and Frédéric Loulergue²

¹ CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

² Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France

Abstract

With the wide expansion of multiprocessor architectures, the analysis and reasoning for programs under weak memory models has become an important concern. This work presents an original constraint solver for detecting program behaviors respecting a particular memory model. It is implemented in Prolog using CHR (Constraint Handling Rules). The CHR formalism provides a convenient generic solution for specifying memory models, that benefits from the existing optimized implementations of CHR and can be easily extended to new models. We briefly present the solver design, illustrate the encoding of memory model constraints in CHR and discuss the benefits and limitations of the proposal.

1 Introduction

Concurrent programs are hard to design and implement, especially when running on multiprocessor architectures. Multiprocessors implement weak memory models [2] that allow, for example, instruction reordering or store buffering. Thus multiprocessors exhibit more behaviors than Lamport's *Sequential Consistency* (SC) [9], a theoretical model where the execution of a program corresponds to an interleaving of the different threads.

Recent years have seen many works on formalization of weak memory models, both on hardware and software sides [12, 4, 10, 5, 6, 3, 1]. These formal models give us a way to compute, given a program and a memory model, the set of concrete executions allowed by the model. We can determine for example whether all executions of a program under a certain model are admitted under SC, and therefore, whether we can reason about this program using an interleaving semantics.

We propose a constraint solving based technique that allows to determine all admissible executions of a program under a memory model. This technique relies on Prolog and CHR (Constraint Handling Rules) [8]. A CHR program is a list of rules that act on a store of constraints, by adding or removing constraints in this store. The rules are activated if a combination of constraints in the store match the head of the rule (and satisfy an optional Prolog predicate called the *guard*). In our case, constraints are relations between instructions that are basically related to their order of execution. Rules are used to propagate information and to produce, from a given set of relations, more relations (characterizing the instruction ordering) that were previously unknown in the execution.

*Work partially funded by French ANR (project SOPRANO, grant ANR-14-CE28-0020).

			<pre> 1 :- include(sc). % :-include(tso).% for SC or TSO 2 3 program_p0(Vars, [Thread0,Thread1]) :- 4 Vars = [x , y], 5 Thread0 = [(st,x,1), (ld,y,R0)], 6 Thread1 = [(st,y,1), (ld,x,R1)]. 7 program_p1(Vars, [Thread0,Thread1]) :- 8 Vars = [x , y], 9 Thread0 = [(st,x,1), f(any,any), (ld,y,R0)], 10 Thread1 = [(st,y,1), f(any,any), (ld,x,R1)]. 11 12 ?- program_p0(Vars, P), apply_model(Vars, P). </pre>												
p_0 :	<table style="border-collapse: collapse;"> <tr> <td style="padding-right: 10px;">Thread 0</td> <td style="border-left: 1px solid black; border-right: 1px solid black; padding: 0 5px;"> </td> <td style="padding-left: 10px;">Thread 1</td> </tr> <tr> <td style="padding-right: 10px;">$(i_{00}) \mathbf{x} = 1;$</td> <td style="border-left: 1px solid black; border-right: 1px solid black;"></td> <td style="padding-left: 10px;">$(i_{10}) \mathbf{y} = 1;$</td> </tr> <tr> <td style="padding-right: 10px;">$(i_{01}) \mathbf{r}_0 = \mathbf{y};$</td> <td style="border-left: 1px solid black; border-right: 1px solid black;"></td> <td style="padding-left: 10px;">$(i_{11}) \mathbf{r}_1 = \mathbf{x};$</td> </tr> </table>	Thread 0		Thread 1	$(i_{00}) \mathbf{x} = 1;$		$(i_{10}) \mathbf{y} = 1;$	$(i_{01}) \mathbf{r}_0 = \mathbf{y};$		$(i_{11}) \mathbf{r}_1 = \mathbf{x};$					
Thread 0		Thread 1													
$(i_{00}) \mathbf{x} = 1;$		$(i_{10}) \mathbf{y} = 1;$													
$(i_{01}) \mathbf{r}_0 = \mathbf{y};$		$(i_{11}) \mathbf{r}_1 = \mathbf{x};$													
p_1 :	<table style="border-collapse: collapse;"> <tr> <td style="padding-right: 10px;">Thread 0</td> <td style="border-left: 1px solid black; border-right: 1px solid black; padding: 0 5px;"> </td> <td style="padding-left: 10px;">Thread 1</td> </tr> <tr> <td style="padding-right: 10px;">$(i'_{00}) \mathbf{x} = 1;$</td> <td style="border-left: 1px solid black; border-right: 1px solid black;"></td> <td style="padding-left: 10px;">$(i'_{10}) \mathbf{y} = 1;$</td> </tr> <tr> <td style="padding-right: 10px;">$(i'_{01}) \mathbf{fence};$</td> <td style="border-left: 1px solid black; border-right: 1px solid black;"></td> <td style="padding-left: 10px;">$(i'_{11}) \mathbf{fence};$</td> </tr> <tr> <td style="padding-right: 10px;">$(i'_{02}) \mathbf{r}_0 = \mathbf{y};$</td> <td style="border-left: 1px solid black; border-right: 1px solid black;"></td> <td style="padding-left: 10px;">$(i'_{12}) \mathbf{r}_1 = \mathbf{x};$</td> </tr> </table>	Thread 0		Thread 1	$(i'_{00}) \mathbf{x} = 1;$		$(i'_{10}) \mathbf{y} = 1;$	$(i'_{01}) \mathbf{fence};$		$(i'_{11}) \mathbf{fence};$	$(i'_{02}) \mathbf{r}_0 = \mathbf{y};$		$(i'_{12}) \mathbf{r}_1 = \mathbf{x};$		
Thread 0		Thread 1													
$(i'_{00}) \mathbf{x} = 1;$		$(i'_{10}) \mathbf{y} = 1;$													
$(i'_{01}) \mathbf{fence};$		$(i'_{11}) \mathbf{fence};$													
$(i'_{02}) \mathbf{r}_0 = \mathbf{y};$		$(i'_{12}) \mathbf{r}_1 = \mathbf{x};$													

Figure 1: (a) Two programs p_0 and p_1 , and (b) their implementations with a solving request.

The contributions of this paper include a novel CHR-based technique for detection of admissible behaviors under memory models and a constraint solver prototype¹ implementing this technique for SC, TSO and PSO models. We discuss the benefits and limitations of this approach, and show how the use of CHR allows us to define our model in a concise and intuitive way, taking advantage of a well-established specification and solving mechanism.

Outline. Section 2 presents weak memory models vs. SC model. Section 3 presents the solver, explains its functionality and optimizations, and points out some soundness and performance aspects. Section 4 discusses its benefits and limitations, and gives future work.

2 Weak Memory Models

A *memory model* describes how a program can interact with memory during its execution. Technical manuals of processors are often vague, and sometimes erroneous. Formal descriptions are necessary, but must be as abstract as possible to ease reasoning about them.

In a parallel context, Lamport [9] proposed *sequential consistency* (SC) where the semantics of the parallel composition of two programs is given by the interleaving of the executions of each program. For example, if we consider that two memory locations x and y initially contain 0 and that r_i are processor registers, the program p_0 in Fig. 1a can only give one of the three possible final results: either $r_0 = 0$ and $r_1 = 1$, or $r_0 = 1$ and $r_1 = 1$, or $r_0 = 1$ and $r_1 = 0$. The last result is obtained by the interleaving $(i_{10})(i_{11})(i_{00})(i_{01})$.

However, for the sake of performance, no current multiprocessor provides a sequentially consistent memory model. Models are *relaxed* or *weak* with respect to SC. In most processors, it is also possible to obtain the result $r_0 = 0$ and $r_1 = 0$, not justifiable by an interleaving. To avoid this behavior, one can use *memory barriers* or *fences* that forbid reorderings through them (see program p_1 of Fig. 1a).

There exist operational models for some weak memory models, for example [6]. However this kind of approaches is not really appropriate to capture the various existing memory models of processors and programming languages. Most approaches are based on constraints expressed as partial orders on events. In the case of memory models, the events of interest are related to memory: *read* (*load*) from, and *write* (*store*) to a memory location, as well as memory barriers.

Constraint-based approach have already been used to model memory models, see for example [11]. In this work, we propose a different approach using the particular case of CHR. Following [3], we use several relations to formalize memory models. The PO relation, for “program-order”, simply expresses that an instruction appears before another one in the list of instructions defining a thread. The CO relation, for “coherency-order”, is a per location total

¹Available at <http://frederic.loulergue.eu/CHR/wmm.tar.gz>.

order of the write actions. The RF relation, for “read-from”, determines which write operation assigned the value at a location before this value is read from this location: it associates a unique write to each load. The FR relation, for “from-read”, relates each load operation with each of the write operations (if any) that overwrite the value after it was read by the load.

3 The Solver

3.1 Overview

A (candidate) execution can be determined by a total ordering (permutation) of memory stores to each location (CO), and an association of a unique store to each load (RF). (They can conflict PO, i.e. the initial order of instructions in threads.) Relations between instructions are modelled by CHR constraints. To compute admissible executions of a program under a given model, we generate all candidate executions (represented by the aforementioned relations), and filter them on-the-fly using CHR rules defined for the model. The set of candidate executions is easily generated (in our case, by a Prolog program using backtrack) by enumerating all possibilities of these relations. Without additional constraints, this set contains the executions authorized by a very weak generic model [3] allowing lots of intuitively incoherent behaviors (e.g. where, within the same thread, an old value that has been overwritten can still be read from the location after the overwriting operation).

To keep only executions allowed by the target memory model, we define CHR rules to filter out cases where the relations between instructions of a given execution are inadmissible. The principle is to compute the transitive closure of certain ordering relations (depending on the model) that the execution must satisfy according to the model. If the computed transitive closure exhibits a cycle, then, according to the model, some program instruction has to be executed strictly before itself, which is incoherent: the execution is not allowed. If this closure has no cycles, the ordering relations are admissible and the execution is allowed by the model. To efficiently use CHRs, we declare all CHR rules needed by the model before the Prolog program, so that they detect incoherent executions as early as possible during their generation.

Language. An input program is modelled by the list of declared variables and the list of lists of instructions of its threads. The implementation of p_0 and p_1 in Prolog is shown in Fig. 1b, lines 4–6, 8–10. The considered instructions are either a memory operation (load or store) or a fence. A load (resp. a store) is a tuple (ld, loc, v) (resp. (st, loc, v)) where loc is the read (resp. written) location and v its value. A fence, written $f(OP1, OP2)$, takes as parameters the types of instructions whose order we want to force. For example, by writing $f(st, ld)$, we state that any store preceding this fence is ordered strictly before any load following the fence. The notation any expresses “any type of operation”. For example, $f(ld, any)$ states that all loads preceding the fence are ordered before any instruction that follows it.

3.2 The Generic Model

A first step is to take the list of instructions of each thread and to enrich them by the thread identifier and instruction index in the list. As we perform this operation, we also create CHR constraints $i/5$, $fence/4$ for memory operations and fences of the form $i(n_thread, n_inst, op, loc, v)$ and $fence(n_thread, n_inst, t_op1, t_op2)$. In this model we consider PO, CO, RF, FR and barrier relations, defined by CHR constraints $po/2$, $co/2$, $rf/2$, $fr/2$, $barrier/2$. We extract po relation by reading the list of instructions of each thread and setting po for two consecutive memory operations in it. For example,

```

1 :- chr_constraint rel/3, trace/3, cycle/2.
2 trace(R,Begin,End) \ trace(R,Begin,End) <=> true.
3 trace(R,Begin,End), rel(R,End,Begin) <=> cycle(R,Begin).
4 trace(R,Begin,End), rel(R,End,Next) ==> inf(Begin,Next) | trace(R,Begin,Next).
5 rel(R,Begin,End) ==> inf(Begin,End) | trace(R,Begin,End).

```

Figure 2: Cycle detection (file `cycle.pl`).

the constraint $po(i(0,0,st,x,1), i(0,1,ld,y,R0))$ will be generated to represent the program order of the two instructions i_{00} and i_{01} of thread 0 of p_0 in Fig. 1a. We also define ipo , the transitive closure of po .

The relations CO and RF are generated by enumerating all possible solutions in a straightforward way. The CO relation is obtained after generating a permutation of all stores to each memory location (with an additional store at the beginning to write *undefined*), and setting the constraints $co(i1,i2)$ for any two consecutive stores $(\dots, i1, i2, \dots)$ in the permutation. The RF relation associates each load to a possible origin store. For the example of execution of Sec. 2, returning $r_0 = 1$ and $r_1 = 0$, the constraints are $rf(i(1,0,st,y,1), i(0,1,ld,y,1))$ and, as we add a store to *undefined* for initialization, $rf(i(-1,-1,st,x,undefined), i(1,1,ld,x,undefined))$. Notice that the read values (here, Prolog variables $R0, R1$) are unified with the written ones. The -1 value of n_thread , n_inst indicates the initial definition of the program memory state.

The relation FR is computed by two rules: $rf(ST,LD), co(ST, ST2) ==> fr(LD,ST2)$ and $fr(LD,ST), co(ST,ST2) ==> fr(LD,ST2)$. The first one means “if there is a RF-relation between ST and LD, and there is a CO-relation between ST and ST2, then add a FR-relation between LD and ST2”. The second adds the subsequent overwritings, if any.

Finally, we have some CHR rules that, taking the set of instruction constraints and the set of fence constraints, produce barrier constraints that force order on instructions. One example for p_1 in Fig. 1a is $barrier(i(0,0,st,x,1), i(0,2,ld,y,R0))$.

3.3 Cycle Detection

Basically a memory model is defined by the relations between instructions that are allowed by the model. As these relations between two instructions express which one appears before the other, we can transitively produce all chains of dependencies. If a chain is a cycle, the execution exhibits an incoherence, since it means that an instruction happens before itself. The detection of a cycle is done by the CHR code of Fig. 2.

We use three CHR constraints. Constraint $rel(R,Begin,End)$ states that instructions $Begin, End$ are related by relation R . The transitive closure $trace(R,Begin,End)$ means that $Begin, End$ are transitively related by a trace, that is, a chain of relations R , while $cycle(R,Begin)$ indicates that a cycle is found from $Begin$ to itself.

The first rule (line 2) is a *simpagation* rule, written $A \setminus B <=> C$. The meaning is “if there exist two constraints A and B , add C , keep A and remove B ”. So here, the goal is to remove duplicate traces with the same origin and end, as they will generate the same final traces.

The *simplification* rule at line 3 expresses that, if we have a trace from $Begin$ to End , and we find a relation between End and $Begin$, we have to replace these two constraints by a new one that indicates the existence of a cycle starting from $Begin$.

The third rule produces the transitive closure by adding longer traces whenever it finds a relation to continue an existing trace, unless it leads to a cycle since the cycle detection is fired

```

1 :- include(generic_model).
2 :- include(cycle).
3 :- chr_constraint sc/2.
4
5 sc(I0,I1) ==> rel(sc, I0, I1).
6
7 sc(I0,I1), sc(I0,I1) <=> sc(I0,I1).
8 po(I0,I1) ==> sc(I0,I1).
9 co(I0,I1) ==> sc(I0,I1).
10 rf(I0,I1) ==> sc(I0,I1).
11 fr(I0,I1) ==> sc(I0,I1).
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Figure 3: Solver files for memory models (a) SC and (b) TSO

by the previous rule. The order of rules is thus essential for performance. To optimize the search further and to limit the quantity of traces we work on, we consider an arbitrary total order inf on instructions, and we add an instruction to a growing trace only if it is strictly greater (w.r.t. inf) than the origin of the trace. Indeed, since we compute traces from every instruction at the same time, to detect any cycle it is sufficient to compute traces going only through instructions strictly greater than the trace origin (cf. Sec. 3.5). We ensure this behavior by the CHR syntax $R \implies \text{Guard} \mid N$ which says “if we match R , add N only if Guard is true”.

The fourth rule selects every known relation between two instructions and adds it as a cycle candidate trace (using the same optimization by inf as above).

If we wish to keep only allowed executions, we can optimize the constraint filtering even further and add another rule $\text{cycle}(_, _) \iff \text{false}$ that fails whenever a cycle is detected. Thus, inadmissible executions will be rejected as soon as detected.

3.4 Formalization of a Memory Model

To formalize a model we first identify the relations that it preserves and we create a new relation that we name according to the model acronym. Each occurrence of a preserved relation will create a new constraint with this name. The goal is then to produce the transitive closure and to launch the detection of cycles for this constraint.

For example, SC preserves the relations PO, CO, RF and FR and it does not care about barriers (since instructions are necessarily kept in order due to PO). We will name the new relation sc . Fig. 3a shows how we model it with CHR. Line 5 launches the computation of the transitive closure. Basically, every time we find a constraint $\text{sc}(A, B)$ we add a new constraint $\text{rel}(\text{sc}, A, B)$. The cycle detection is provided by the inclusion of cycle.pl , line 2. Cycle rules and rel rule are added before any other rule about SC, to ensure the earliest detection of cycles. Lines 7–11 show how to state that sc preserves other relations. Every time such a relation is met, we add a new sc -relation.

Another interesting case is when the target model is more relaxed, that is, when the model preserves only some relations of the generic model. For example the TSO model will relax two kinds of relations: program order when the instructions are a store followed by a load, and read-from when the two instructions are in the same thread. The reordering propagates transitively: multiple stores can be reordered after consecutive loads. If the developer wants to restore these relations, they will have to add fences. This model is illustrated in Fig. 3b.

We add a relation named `ppo` (“preserved-program-order”). Every relation `ipo` will generate a relation `ppo` except the ones mentioned before. So we add the rule on line 5, that replaces every constraint `ppo(store, load)` by “true”, which means that we just remove it, and a rule (line 6) that will generate `ppo` from `ipo`. We add another relation named `rfe` (“read-from-external”). The associated rule (lines 8-9) builds `rfe`-relations from `rf` by only adding those that concern instructions in different threads. Finally, we create the `tso` relations as for SC. Again, the order of rules here is essential for both soundness and performance (cf. Sec. 3.5).

Most weak memory models respect a coherency between communications (CO, RF, FR) and PO per location. In essence, it restores the uni-processor coherency (e.g. it forbids to read an old value from an overwritten location, that is allowed by the generic model, as mentioned in Sec. 3.1). It is formalized in the included `uniproc` file (not detailed here), which includes and relies on `cycle`, so we do not have to include it again.

The implementation of the PSO model is quite straightforward once TSO is implemented. It consists in weakening the preserved-program-order a bit more by removing all `ipo`-pairs starting with a store instead of only removing store-load pairs. Concretely, we replace the rule line 5 in the TSO model by the following one: `ppo(i(_,_,st,_,_), _) <=> true`.

Applying the solver to programs. The application of the solver for a given model to a program is performed automatically as illustrated in Fig 1b for program p_0 described in Sec. 2. We include the target model, that already includes the generic model with all other necessary definitions. The program is loaded using two variables (variables involved in the program and lists of instructions of threads). Applying the target model solver to the program (line 13) will generate all candidate executions, and the CHR rules of the model will determine if each execution is allowed by the model or not.

In this case the SC model detects 3 admissible executions for both programs p_0 and p_1 . The TSO model allows 4 admissible executions for p_0 , while for p_1 , as we have fences, some relations will be restored and TSO will allow only 3 executions (exactly as SC). TSO allows more executions than SC on p_0 since ST/LD pairs of instructions can be reordered (that can lead to the result $r_0 = 0$ and $r_1 = 0$). In p_1 , we restore the sequentially consistent behavior of p_0 by adding fences, preventing reordering even in the TSO model.

3.5 Discussion on Soundness and Performance

While using CHR, the order of rules can be essential for both soundness and performance. We emphasize two particular points regarding the order of rules in the proposed solver: cycle detection and computation of preserved-program-order relation `ppo`.

In cycle detection (cf. Fig. 2), we want to ensure that the solver does not miss any traces and avoids to consider equivalent traces when possible. First, as mentioned in Sec. 3.3, the rule at line 2 in Fig. 2 removes equivalent traces that have exactly the same origin and end points. As it is the first rule, we will not waste time by using the following rules for two equivalent traces since we will keep only one of them. Second, trace generation starts simultaneously from all known relations `rel(R, Begin, End)` where instruction `Begin` is less than `End`, while other instructions are added at the end of a trace only if they are greater than its origin (lines 4-5). So, every cycle has a minimal instruction from which we can find this cycle as a trace from an origin through greater instructions followed by a return to the origin, cf. line 3. (Notice that we still need to consider subtraces of the same cycle when they start from other nodes.) Third, the addition of an instruction to a trace (line 4) is done by adding a new constraint, without removing the “old” ones that can still be necessary to generate other grown traces, so we do not miss paths.

When defining a relation such as preserved-program-order ppo (cf. Sec. 3.4), there are two ways to proceed: to define preserved relations or to define removed relations. The first way is always sound (as we only add information), while the second one requires more care. In particular, it is important to place the rule which removes relaxed relations before the rules that generate the constraints for cycle detection (see e.g. lines 5–6, 11, 13 in Fig. 3b). Otherwise, these rules would generate constraints for cycle detection before the relaxed relation is actually removed from the store and could thus forbid allowed executions.

Experiments. We tested our solver for different models against 18 examples² provided for the implementation of a dedicated tool Herd [3]. On these small (but representative) examples, our solver returns the same allowed executions as those found by Herd.

Execution time on these examples being too fast, we also compare performances of our solver with Herd on some examples involving a series of (possibly wrong) message passing. Experiments have been performed on an Intel Core i7-4800MQ, 4 cores, 2.7GHz, 16Go RAM. For example, the following code involves three message passing:

```

1 mp3(V, [T0,T1,T2]) :-
2   V = [ x, m ] ,
3   T0 = [ (st,x,1), (st,m,1), (ld,m,M0), (ld,x,X0) ],
4   T1 = [ (ld,m,M1), (ld,x,X1), (st,x,2), (st,m,2) ],
5   T2 = [ (ld,m,M2), (ld,x,X2), (st,x,3), (st,m,3) ].

```

In such an example, the typical question is: if we get $M0 = 3$, $M1 = 1$ and $M2 = 2$, do we get $X0 = 3$, $X1 = 1$ and $X2 = 2$?

As it is composed of multiple writes and reads to the same locations, the combinatorial explosion is very fast. For each model, we indicate the number of allowed executions and the time needed to compute them with our CHR-based solver and with Herd. The timeout is fixed to 1 hour. The number of executions indicated for the generic model corresponds to the number of candidate executions. We present here the results for 3 and 4 message passings (denoted 3MP and 4MP).

In this benchmark, our solver is configured to immediately reject forbidden executions as soon as they are detected, while Herd does not perform such early rejections. When combinatorial explosion becomes really big, our solver computes allowed executions faster than Herd thanks to an early pruning of the search tree.

Model	Data	3MP	4MP
SC	#exec	678	81 882
	CHR	3.3s	747s
	Herd	5.5s	> 1h
TSO	#exec	800	96 498
	CHR	3.2s	752s
	Herd	4.1s	> 1h
PSO	#exec	2 258	516 030
	CHR	6.4s	2796s
	Herd	3.8s	> 1h
Generic	#exec	147 436	255 000 000
	CHR	3.3s	> 1h
	Herd	1.2s	1405s

4 Conclusion and Future Work

We have presented an original CHR-based solver for detection of admissible executions of a given program w.r.t. a given memory model, and illustrated it for SC, TSO and PSO models. It is suitable for a rigorous exhaustive analysis of program executions of small programs that becomes intractable for bigger ones due to the combinatorial explosion of their number.

²Examples available at <http://virginia.cs.ucl.ac.uk/herd/> (record “armed cats”).

We think that seeing memory models as constraints over executions is well adapted. The design of such a solver is convenient and pragmatic. The generation of basic executions and cycle detection relies on a few optimizations in order to be more efficient and to ensure on-the-fly filtering of constraints. The proposed approach makes the definition of specific models from the generic one very practical and relatively straightforward. In particular, it is not very hard when the model becomes more complicated, as for models like ARM for example. CHR provides an easy way to express constraints about execution of programs, they have also been used for detection of incorrect behaviors in imperative program analysis in [7].

Moreover, the use of a well established mechanism of constraint specification and solving, here Prolog and CHR, brings the benefit of years of optimization and debugging to handle our problem without having to re-develop constraint resolution. We do not claim that our implementation of this problem is most efficient. Dedicated tools like [3] could be faster since they are specialized for this precise problem and can implement a solving engine without being as generic as CHR. But such dedicated tools are harder to develop as they potentially require a new optimized code that has to be carefully developed and debugged.

In future work, we plan to extend the solver to other models, like ARM. It would be interesting to support other kinds of instructions (e.g. binary operations) to handle all kinds of data or address dependencies often used for synchronization in ARM. Another direction would be to experiment on different programs of various sizes to produce precise benchmarks, in order to compare the solver to dedicated tools, as well as to further optimize it.

Acknowledgment. The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Defence. Thanks to the anonymous referees for their helpful comments.

References

- [1] T. Abe and T. Maeda. Optimization of a general model checking framework for various memory consistency models. In *PGAS 2014*.
- [2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.
- [3] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Prog. Lang. Syst.*, 2014.
- [4] Arvind and J. Maessen. Memory model = instruction reordering + store atomicity. In *ISCA 2006*.
- [5] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI 2008*.
- [6] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *POPL 2009*.
- [7] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification using constraint handling rules and array constraint generalizations. In *VPT 2014, co-located with CAV 2014*, pages 3–18.
- [8] T. Frühwirth. Theory and practice of constraint handling rules. *J. Logic. Program.*, 1998.
- [9] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program. *IEEE Trans. Comput.*, 1979.
- [10] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPoPP*, pages 161–172. ACM, 2007.
- [11] E. Torlak, M. Vaziri, and J. Dolby. Memsat: checking axiomatic specifications of memory models. In *PLDI*, pages 341–350, 2010.
- [12] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: An operational memory model specification framework with integrated model checking capability. *Concurr. Comput. : Pract. Exper.*, 17(5-6):465–487, 2005.