JmmSolve: a generative reference implementation of CCM Machines

Tom Schrijvers *Bart Demoen*

Report CW 379, January 2004



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

JmmSolve: a generative reference implementation of CCM Machines

Tom Schrijvers *Bart Demoen*

Report CW 379, January 2004

Department of Computer Science, K.U.Leuven

Abstract

In this paper we report on JmmSolve, a solver implementation of CCM machines, a framework for Java memory models. It illustrates the generative nature of the memory model framework in that it is able to generate all valid read-write linkings.

Keywords : Java, memory model, constraint programming, CHR
CR Subject Classification : D.3.1, D.3.3

JmmSolve: a generative reference implementation of CCM Machines

Tom Schrijvers, Bart Demoen

March 9, 2004

Abstract

In this paper we report on JmmSolve, a solver implementation of CCM machines, a framework for Java memory models. It illustrates the generative nature of the memory model framework in that it is able to generate all valid read-write linkings.

1 Introduction

The object of the Concurrent Constraint-based Memory (CCM) Machines framework is to provide a fully specified concurrent memory model for the Java language [8].

While the framework follows the common requirements of such a model, it also aims to satisfy an additional requirement:

M6. The model should be generative in nature. This means that given a program it should be possible to use the model to generate all possible execution sequences of the program.

It is the latter requirement that is the object of this report: we present a generative reference implementation for CCM Machines.

Since the CCM machines are specified in terms of constraints, we have chosen to implement the generative solver with Constraint Handling Rules. Constraint Handling Rules, or CHR for short, are a high level rule-based language (see [2]) that performs a bottom-up fixpoint computation. While CHR has currently many applications, it has been designed for writing constraint solvers in particular. Indeed, its compact and declarative syntax is excellent for prototyping succinct constraint solvers. Although its performance is not on par with constraint solvers written in lower-level languages,

its flexibility and ease of use do favor a wider use, especially for prototype implementations like the one presented here: JmmSolve.

CHR is not a self-contained but an embedded language. Although versions of CHR for Haskell and Java do exist, Prolog is its original and natural host. Just as the traditional combination of Constraint and Logic Programming [7], the combination of CHR with Logic Programming seems to be the most powerful and expressive combination. JmmSolve has been implemented in SWI-Prolog [9] because it is a popular free Prolog system and we are the authors of its CHR system. However, a port to another Prolog system with CHR would be trivial.

The report is structured as follows. First, in Section 2 we briefly review the major concepts of CHR. For a more thourough overview we refer the reader to [3]. The implementational aspects of the reference implemenation are discussed in Section 3. Next, we suggest several future extensions and possibilities for efficiency improvements in Section 4. Finally, in Section 5 we conclude.

2 Constraint Handling Rules by Example

The set of labeled[1] constraint handling rules below defines a less-than-or-equal constraint (`leq/2`) over numbers. The rules illustrate several syntactical features of CHR.

```
 reflexivity @  leq(X,X) <=> true.
     integer @  leq(X,Y) <=> number(X), number(Y) | X =< Y.
antisymmetry @  leq(X,Y) , leq(Y,X) <=> X = Y.
setsemantics @  leq(X,Y) \ leq(X,Y) <=> true.
transitivity @  leq(X,Y) , leq(Y,Z) ==> leq(X,Z).
```

The first, second and third rule are simplification rules, indicated by the double arrow. To the left of the arrow is the head of a rule. A simplification rule has the meaning that the constraints in the head can be simplified to the Prolog goal in the body, `true` for the first rule. Instead of the full unification used in the head of a Prolog clause, matching is used in the head of a CHR rule. The first rule means that the leq relation is reflexive, and hence that `leq(X,X)` is trivially satisfied and can be discarded.

The second rule shows that a rule can also contain a guard, after the arrow and before the vertical bar. In this case the guard is `number(X),`

[1] The name before the @ is optional and has no semantics meaning.

`number(Y)`. The body of the rule is only executed for constraints that match the head and satisfy the guard. The guard can be any Prolog goal that does not bind variables appearing in the head to each other or further instantiate them. Rule two replaces the constraint with a simple Prolog inequality check if the arguments are bound to numbers.

The third rule illustrates that the head of a rule can contain a conjunction of multiple constraints. It formulates the antisymmetry property of the leq constraint.

The fifth rule with the `==>` is a propagation rule. The body of the rule is executed once for every matching combination of constraints in the head. It does not remove the head constraints like the simplification rules do.

The fourth rule is a "simpagation" rule. It has the same meaning as a simplification rule where the constraints before the backslash would be posed again in the body. However it is more efficient in that it never removes those head constraints and does not unnecessarily trigger rules in that way. In the leq constraint definition its role is to declare the set semantics of the constraint, i.e. the number of copies of a constraint is not important and hence it is more efficient to keep only one.

Operationally, when a constraint is posed the rules are tried in order. For a multi-headed rule, the additional constraints are looked for in the constraint store. The outcome is that the posed constraint is either simplified away or reaches the end of the rules and gets suspended in the constraint store. In its suspended state it can either be used as an additional constraint for a multi-headed rule or wait until it gets triggered. Triggering of a suspended constraint occurs when any variable in the constraint gets bound. The constraint then tries to match all rules in order again.

3 Implementation of JmmSolve

The object of JmmSolve is to generate for a particular CMM machine instantiation all valid constraint stores for multi-threaded programs. Of particular concern is the generation of all valid read-write linkings and the corresponding values that are read and written. In addition, the necessary additional orderings have to be generated according to the underlying memory model.

The full code of JmmSolve is listed in Appendix A and a program encoding example can be found in Appendix B.

3.1 Input of the solver

Basically, the input of the solver is a multi-threaded program. In particular the following information is supplied:

- The *initial value* of all shared and thread-local variables.
- The *events and actionsets* of the different threads.
  An event is either a read of a variable or the unconditional or conditional write of a variable. In addition to the events, constraints are given that relate the values read and/or written. In addition, for local variables all reads and writes within the same actionset need to be ordered.
- The partial *order between the different actionsets of a single thread.* Since actionsets correspond with groups of events that are shipped toghether to the constraint store, the order of these sets sent by a particular thread should be given.

3.2 Generated ordering constraints

With each event E a unique ordering variable O_E of an unspecified domain is associated in the CCM machine. A partial order `(<<)/2` over these ordering variable represents the partial ordering of the corresponding events.

In JmmSolve the ordering variables have been chosen to be unique atoms and the `(<<)/2` constraint expresses ordering between them. Appropriate CHR constraints to capture the semantics of the ordering relation are:

```
antireflexive @ O << O <=> fail.
antisymmetric @ Ox << Oy, Oy << Ox <=> fail.
transitive    @ Ox << Oy, Oy << Oz ==> Ox << Oz.
```

In addition, the underlying memory model of the CCM machine may require some orderings on events that have not been specified in the input program.

For example, the Location Consistency model requires that all events on the same memory line are ordered. JmmSolve, as a generative solver, generates all possible orderings between such events by alternatively ordering one event before the other and vice versa.

3.3 Read-write linkings

The CMM Machines require a valid linking of all read events with a write event. JmmSolve takes the *generate & test* approach by generating all possible linkings and checking whether they are valid.

Here are the rules related to the linking of read events to writes events:

- A linking of a read to an unconditional write is valid, if the write is a maximal event occuring before the read or the read is not order related to the write.
- A linking of a read to a conditional write results in a conditional read. If the condition of the write is met than the read is linked to that write. Otherwise, the read is linked to a different write which is visible to the read, given that the unconditional write is not active.

  More on the constraint representation of a conditional read can be found in Section 3.4.

In addition a valid linking of reads to writes forces all read and written values to be unique. This is called the *Unique Solution Criterion*; its implementation is disucssed in detail in Section 3.5.

3.4 Conditional read representation

Reads that are linked to conditional writes, are called conditional reads in [8]. They are denoted with the ternary `?:` operator as for example:

```
Xr = (Cond)?Xw1:Xi
```

In JmmSolve the condition `Cond` is translated into a reified constraint of the integer solver. A reified constraint constrains a boolean variable to be either `true` or `false`, depending on whether the condition is implied or inconsistents with the constraint store. In addition the ternary operator `?:` is translated into an implication-like if-then-else constraint `ite/4` that binds the read variable to one of the other variables depending on the boolean. The following are CHR rules defining the behavior of `ite/4`:

```
ite(true,Xr,Xw1,Xi)  <=> Xr = Xw1.
ite(false,Xr,Xw1,Xi) <=> Xr = Xi.
ite(B,Xr,X,X)        <=> Xr = X.
```

More intelligent rules might be added for particular variable domains.

Consider the following example code:

```
y == 0 initially

r1 = x
if (r1 == 1)
      y = 1;
r2 = y;
```

then the following linking for `Yr` is possible:

```
Yr = (R1 == 1) ? Yw1 : Yi
```

This linking is translated into the following conjunction of JmmSolve constraints:

```
reified_eq(R1,1,Bool), ite(Bool,Yr,Yw1,Yi)
```

3.5 The Unique Solution Criterion

The *Unique Soluation Criterion* dictates that a valid linking forces all variables to take on unique values. In theory, with a complete constraint solver, during linking:

- execution fails if no solution exists
- all variables are grounded if just one solution exists
- unbound variables remain if there are multiple solutions

Hence, it suffices to check after linking whether unbound variables remain, to decide whether a unique solution exists. Unfortunately, in practice the used constraint solvers are not complete and variables may remain while there is no or just one possible value for them.

This can be checked after the linking by labelling the remaining variables. The Prolog all solutions predicate `setof/3` is a simple convenient means to check whether one or more possible labellings exist:

```
label_vars_uniquely(Vars) :-
       setof(Vars,label(Vars),[Valuation]),
       Vars = Valuation.
```

3.6 Expressions over value domains

The values that are read from and written to memory lines are represented by solver variables. The domains of the variables should correspond with the primitive types or object references of Java. For simplicity, they are currently restricted to the integers provided by the underlying Prolog system.

A new integer value can be computed via the standard Java arithmetic functions like `+`, `-`, `*` and so on. In the JmmSolve store not all the values appearing in an expression are known already; they may still be represented as unbound variables. Hence, to capture the relation between the values, they are represented as solver variables of an integer solver and the expression is translated into a constraint of the integer solver.

For example, the statement from Test 8 of [8] contains an arithmetic expression:

```
r2 = 1 + r1 * r1 - r1
```

The values the two threadlocal variables `r1` and `r2` are represented by two integer solver variables `R1` and `R2`. The statement is translated into the following integer constraint:

```
R2 eq 1 + R1 * R1 - R1
```

where `eq/2` is the equality constraint of the integer solver. If an expression does not appear on the right-hand side of an equality constraint, the program may be transformed to bring them into this form. Simply, every expression that does not appear on the right-hand side of an equality constraint is replaced by a fresh variables. For each such fresh variable, a constraint is added to equate it to the replaced expression. In this way we do not need to be concerned with expressions appearing anywhere else than in equality constraints.

4 Future work

4.1 Full semantics support

Currently, there is no support for the locking and unlocking on a particular variable. The Location Consistency and Happens Before memory models both impose a number of restrictions on the ordering of locking operations. These should also be generated by the JmmSolve solver.

This will be investigated as soon as the semantical aspects of locking for CCM machines, have been fixed. The same holds for other additional concepts of CCM machines.

4.2 Expressions

For the JmmSolve prototype, only a limited integer solver has been implemented. Additional types of variables such as boolean and float may be supported as well. In addition solvers that more faithfully observe the semantics of these types according to the Java language specification [5] should be considered. For example, the integer solver should deal with overflow of the four byte representation of Java according to the specified modulo semantics.

4.3 Source parser

For experimentation purposes, it would be quite useful to have an automatic translation of source programs into events and actionsets of constraints. The parser for such a task could be written with the Definite Clause Grammar extension of Prolog. The translation is mostly straightforward, although several complications do occur, such as:

- translation of an if-then-else into two conditional blocks
- translation of an array read or write with a variable index into multiple conditional events
- dealing with while-loops

4.4 Suggestions for Efficiency Improvements

Several approaches to further improve the efficiency of JmmSolve are possible:

Efficient Expression Solver The current solver for expressions is a minimal one implemented in CHR. It suspends evaluation until all variables are ground. Hence, the search space that the labelling has to explore is not reduced by constraints depending on expressions.

It is however rather easy to replace the current prototype CHR expression solver with a more efficient solver, e.g. the `clp(FD)` solver of SICStus [6] or the `ic` solver of Eclipse [1]. Such a more efficient solver will propagate and simplify constraints and ground variables before labelling. This can

greatly reduce the search space during labelling and hence have a considerable impact on performance.

Efficient Memory Model Solver Currently JmmSolve uses CHR rules for imposing the constraints of the ordering model. This CHR implementation may be replaced by a lower-level implementation using global and attributed variables. Such an implementation would be able to exploit information on types, modes and order of addition to improve on the joins of multi-headed CHR rules. Information could be stored in specific, possibly global, datastructures for faster lookup, for example.

Alternatively, a typed and moded Prolog-variant with an efficient CHR-system that exploits these properties, may be consdired. Currently, the authors are aware of one such system, namely HAL [4]. Unfortunately, this system is currently not available for public use.

5 Conclusion

We have shown that the memory model of the CCM machines is indeed a generative model by implementing a solver-generator: JmmSolve. This solver allows Java compiler programmers to experiment with the model in several ways:

- check the validity of a proposed compiler optimization on example programs
- look for potential allowed optimisations by condering the valid linkings of programs

JmmSolve still requires much more experimental evaluation to discover its performance hotspots. These should then guide improvements based on the presented suggestions.

The current implementation of JmmSolve is available for download at `http://www.cs.kuleuven.ac.be/~toms/jmmsolve/`.

References

[1] Aggoun, Abderrahamane and Chan, David and Dufresne, Pierre and Falvey, Eamon and Grant, Hugh and Harvey, Warwick and Herold, Alexander and Macartney Geoffrey and Meier, Micha and Miller, David and

Mudambi, Shyam and Novello, Stefano and Perez, Bruno and van Rossum, Emmanuel and Schimpf, Joachim and Shen, Kish and Tsahageas, Periklis Andreas and de Villeneuve, Dominique Henry. *ECL^{i}PS^e User Manual. Release 5.3.* European Computer-Industry Research Centre, Munich and Centre for Planning and Resource Control, London, 2001.

[2] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 90–107. Springer Verlag, March 1995.

[3] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[4] M. García de la Banda, B. Demoen, K. Marriott, and P. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 47–66. Springer-Verlag, 2002.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley Longman, Inc., 1996.

[6] Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* PO Box 1263, SE-164 29 Kista, Sweden, October 2003.

[7] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM Press, 1987.

[8] V. Saraswat. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models (Preliminary Report). Technical report, IBM, March 2004.

[9] J. Wielemaker. SWI-Prolog's Home. http://www.swi-prolog.org/.

Appendix A: JmmSolve code

The code below is the current prototype implementation of JmmSolve

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% JmmSolve, a reference implementation of CCM Machines
```

```
%
% Author:     Tom Schrijvers
% E-mail:     Tom.Schrijvers@cs.kuleuven.ac.be
% Copyright:  2004, K.U.Leuven
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- use_module(library(lists)).

constraints
        collect_events/1,
        collector/1,
        event/6,
        ite/4,
        link/2,
        reified_eq/3,
        reified_neq/3,
        reified_geq/3,
        expression/2,
        visible_write/3,
        (<<)/2,
        as/1
        .

% semantics of order relation
O << O <=> fail.
Ox << Oy, Oy << Ox <=> fail.
Ox << Oy \ Ox << Oy <=> true.
Ox << Oy, Oy << Oz ==> Ox << Oz.


% reified constraints
reified_geq(X,Y,T) <=> number(X), number(Y) |
                ( X >= Y ->
                        T = true
                ;
                        T = false
                ).

reified_eq(X,X,T) <=> T = true.
```

```
reified_eq(X,Y,T) <=> X \= Y | T = false.

reified_neq(X,X,T) <=> T = false.
reified_neq(X,Y,T) <=> X \= Y | T = true.

% if-then-else constraint implication
ite(V,true,T,E) <=> V = T.
ite(V,false,T,E) <=> V = E.
ite(V,C,B,B) <=> V = B.

% eq/2 with variable on left-hand side and expression on right-hand side
expression(V,E) <=> ground(E) | metacalled_is(E,V).

% CHR pattern to collect all constraint of a certain type
collect_events(Es) ==>
        collector([]).

event(OA,OE,T,L,V,E) \ collector(List) <=>
        \+ memberchk_eq(event(OA,OE,T,L,V,E),List)
        | collector([event(OA,OE,T,L,V,E)|List]).

collect_events(Es), collector(L) <=> Es = L.

% link reads to writes
link_reads(Events) :-
        link_reads(Events,Events).

link_reads([],_).
link_reads([E|Es],Events) :-
        link_read(E,Events),
        link_reads(Es,Events).

link_read(event(_,Or,_,L,Var,Type),Events) :-
        ( Type == read ->
                select(event(_,Ow,_,L,Value,OType),Events,REvents),
                ( OType == write ->
                        visible_write(Ow,Or,[]),
                        link(Or,Ow),
                        Var = Value
                ; OType = write(C) ->
```

```
                        visible_write(Ow,Or,[]),
                        link(Or,ite(C,Ow,OElse)),
                        ite(Var,C,Value,Else),
                        ite_else(Or,L,[Ow],REvents,Else,OElse)
                )
        ;
                true
        ).

ite_else(Or,L,Passives,Events,Else,OElse) :-
        select(event(_,Ow,_,L,Value,Type),Events,REvents),
        is_write(Type),
        visible_write(Ow,Or,Passives),
        ( Type == write ->
                Else = Value,
                OElse = Ow
        ;
                Type = write(C) ->
                ite(Else,C,Value,Rest),
                OElse = ite(C,Ow,OELse1),
                ite_else(Or,L,[Ow|Passives],REvents,Rest,OElse1)
        ).

is_write(write).
is_write(write(_)).

% semantics of visible writes, with a list of passive writes
visible_write(Ow,Or,_), Or << Ow <=> fail.
visible_write(Ow,Or,Ps), Ow << Ox, Ox << Or, event(_,Ox,_,_,_,Type)
        <=> is_write(Type), \+ memberchk(Ox,Ps)
        | fail.

% add actionset to constraint store
actionset(OA,T,Events) :-
        as(OA),
        add_events(Events,OA).

add_events([],_).
add_events([E|Es],OA) :-
        add_event(E,OA),
```

```
        add_events(Es,OA).

add_event(write(T,L,OE,Val),OA) :-
        event(OA,OE,T,L,Val,write).
add_event(read(T,L,OE,Var),OA) :-
        event(OA,OE,T,L,Var,read).
add_event(eq(X,Y),_) :-
        X = Y.
add_event(expression(V,E),_) :-
        expression(V,E).
add_event((Cond -> Write),OA) :-
        Write = write(T,L,Ow,Val),
        add_condition(Cond,TruthValue),
        event(OA,Ow,T,L,Val,write(TruthValue)).
add_event(X << Y,_) :-
        X << Y.

add_condition(geq(X,Y),T) :-
        reified_geq(X,Y,T).
add_condition(eq(X,Y),T) :-
        reified_eq(X,Y,T).
add_condition(neq(X,Y),T) :-
        reified_neq(X,Y,T).

% check for unique labelling
label_uniquely(Events) :-
        term_variables(Events,Vars),
        label_vars_uniquely(Vars).

label_vars_uniquely(Vars) :-
        setof(Vars,label(Vars),Valuations),
        Valuations = [Valuation],
        Vars = Valuation.

label([]).
label([V|Vs]) :-
        ( var(V) ->
                domain(D),
                member(V,D)
        ;
```

```
                true
        ).

domain([-5,-4,-3,-2,-1,0,1,2,3,4,5,42]).

% order on actionsets induces order on events [here for LC model]
as(OAf),as(OAs),OAf << OAs, event(OAf,OEf,_,L,_,_),event(OAs,OEs,_,L,_,_)
        ==> OEf << OEs.

% required ordering relations for LC memory model
event(_,Ox,T,L,_,_), event(_,Oy,T,L,_,_) ==>
        true |
        ( Ox << Oy ; Oy << Ox).
/*
% required ordering relations for HB memory model
event(_,Ox,T,_,_,_), event(_,Oy,T,_,_,_) ==>
        true |
        ( Ox << Oy ; Oy << Ox).
*/
```

Appendix B: Program Encoding Example

The code below illustrates the encoding of programs for JmmSolve. The example program is Test 8 of [8]:

```
init x = 0;
init y = 0;

thread {
  r1 = x;
  r2 = 1 + r1 * r1 - r1;
  y = r2;
} |
thread {
  r3 = y;
  x = r3;
}
```

The actionset encoding of the program looks like:

```
test8(R1,R2,R3) :-
        actionset(oI,init,
                [write(init,x,oxi,Xi),
                 write(init,y,oyi,Yi),
                 eq(Xi,0),
                 eq(Yi,0)
                ]),
        actionset(oA1,1,
                [read(1,x,or1,X1r),
                 write(1,y,oy1,Y1w),
                 eq(X1r,R1),
                 eq(Y1w,R2),
                 expression(R2,1+R1*R1-R1)
                ]),
        actionset(oA2,2,
                [read(1,y,or3,Y2r),
                 write(1,x,ox2,X2w),
                 eq(R3,Y2r),
                 eq(X2,R3)
                ]),
        oI << oA1,
        oI << oA2,
        collect_events(Events),
        link_reads(Events),
        label_uniquely(Events).
```