Using the Causal Calculator with the $\mathcal{C}$ Input Language (Draft)

Norman McCain
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188, USA
mccain@cs.utexas.edu

June 10, 1999

The *Causal Calculator* (also, called *CCALC*) is a model checker for the language of causal theories [McCain and Turner, 1997 and 1998].[1] The $\mathcal{C}$ language [Giunchiglia and Lifschitz, 1998] is a high-level, action description language. Using rewrite rules, propositions in $\mathcal{C}$ can be translated into the language of causal theories. The translation is defined in Appendix B, but otherwise is not discussed in this manual. Our focus here is exclusively on using $\mathcal{C}$ as an input language for query answering and planning with the Causal Calculator. The reader is assumed to be familiar with $\mathcal{C}$ — specifically, the version defined in [Gelfond and Lifschitz, 1998]. Otherwise, the manual is self-contained.

A $\mathcal{C}$ input file consists of declarations, comments, and propositions in $\mathcal{C}$ — or, more often, schemas (with meta-variables) whose instances are propositions in $\mathcal{C}$. The primary declarations are those that describe the signature of the language in which the domain description is written.

The behavior of the Causal Calculator in planning can be summarized as follows. Given a $\mathcal{C}$ input file describing an action domain, the propositions and schemas in the file are translated into laws and schemas of the language of causal theories. The Causal Calculator instantiates the schemas with respect to the language signature — replacing schematic variables by ground terms of the language. The resulting ground theory is translated into propositional logic and transformed into a set of clauses. When a planning problem is posed, a range of times is specified (either implicitly or explicitly) by a set of natural numbers $\{0, 1, \ldots, n\}$ ($n >= 1$). From the previously generated set of clauses, another set of clauses is obtained whose models correspond to the causally possible histories of the domain over the specified range of times. The second set of clauses is combined with clauses describing a possible initial state (and perhaps facts about other times as well) and a set of time-specific goals. Then a model of the resulting set of clauses is sought. If a model is found, a plan is extracted from it, displayed, and optionally verified (see below).[2]

---

[1] Vladimir Lifschitz, Hudson Turner, and Enrico Giunchiglia contributed to the design of the Causal Calculator. Hudson Turner and Armando Tacchella contributed to the implementation.

[2] The Causal Calculator has two modes of operation, called *history* and `transition` modes. The preceding summary describes the operation of the system in transition mode. (Certain variations – including the option

Presently, the user may select either of two satisfiability checkers, (i) *rel_sat*, due to Bayardo and Schrag [1997], or (ii) *sato*, due to Hantao Zhang [1997]. Under special conditions (analyzed for the case of causal theories in [McCain and Turner, 1998]), model-finding can be viewed as planning. The essential idea behind this approach to planning is due to Kautz and Selman [1992, 1996].

The manual is organized as follows. In the next section, we give instructions for running the Causal Calculator. Following this, in Sections 2–4, we illustrate its use in planning with respect to several small domains. In Section 5, we describe the syntax of a $\mathcal{C}$ input file, including the syntax of formulas. Separate subsections describe each of the allowed declarations. In Section 6, we describe the operators and their precedences. We conclude in section 7 by describing a small set of user-defined parameters.

There are four appendices. The first appendix summarizes the syntax. The second appendix contains the so-called *standard $\mathcal{C}$ file*, which customizes the Causal Calculator for use with the $\mathcal{C}$ input language. The third appendix describes and illustrates the use of history mode. The fourth appendix contains a variant of the standard $\mathcal{C}$ file, which is required for history mode.

1 How to Run the Causal Calculator

If you wish to run CCALC, you will need either SICStus Prolog (Version 3) or Gnu Prolog (Version 1) running on a Sun workstation. (Presently, CCALC runs much faster under SICStus Prolog than under Gnu Prolog, so you should use SICStus Prolog if at all possible.) If you have access to this platform, you may wish to download the system file `ccalc.tar.gz` from the web page at the following URL.

`http://net.cs.utexas.edu/users/mccain/cc/`

Downloading and unpacking the system file will create a directory under your current directory named `ccalc`. To proceed, first make `ccalc` your current directory. If you have SICStus Prolog, start it up, and then compile and load the file `ccalc.pl` as follows.

```
SICStus 3.7.1 (SunOS-5.5.1-sparc): Tue Oct 06 13:38:15 MET DST 1998
Licensed to cs.utexas.edu
    | ?- compile('ccalc.pl').
```

After a few seconds Prolog will again prompt you for input.

```
    | ?-
```

If you do not have SICStus Prolog but have Gnu Prolog, then compile the file `ccalcgp.pl` from unix to produce a stand alone executable and run it as follows.

```
    unix% gplc ccalcgp.pl
    unix% ccalcgp
```

of planning by backtracking over a expanding range of times – are illustrated in section 2.) In history mode, the range of times is specified in the input file rather than in individual planning problems, and the final set of clauses (whose models correspond to the causally possible histories of the domain) is obtained directly. Transition mode is appropriate for input languages, such as $\mathcal{C}$, that have a "transition system" semantics – although $\mathcal{C}$ input files may also be run in history mode (see Appendix C).

At this point, you are ready to begin interacting with the Causal Calculator in the manner illustrated in the following sections. To exit Prolog for good, type

```
| ?- halt.
```

Warning: When the Causal Calculator is run, it will overwrite files in the current directory with the following names. (Whether or not `ccsat.sorted` is written out depends on the setting of an user-defined parameter. See Section 7.)

```
ccsat.in0
ccsat.in1
ccsat.in2
ccsat.in
ccsat.out
ccsat.sorted
```

It also may erase and then recreate an external database of rules, which (if it exists) resides in the subdirectory `rule_db`.[3]

The SICStus Prolog manuals can be found at the following URL.

http://www.cling.gu.se/datorinfo/sicstus/

2 The Yale Shooting Domain with Two Guns

In this section, we show how to use the Causal Calculator for query answering and planning. The example uses the $\mathcal{C}$ input file, named `yale2.c` shown in Figure 1. It formalizes the Yale Shooting domain [Hanks and McDermott, 1987] with two guns. The file contains a number of declarations and propositions. The first declaration says to include the file named `C`, which is known as the standard $\mathcal{C}$ file and is shown in Appendix B. The standard $\mathcal{C}$ file includes the macros that translate $\mathcal{C}$ into the internal language of the Causal Calculator and the standard sorts, variables, constants, and causal laws that are useful in formalizing action domains for the Causal Calculator. A portion of the standard $\mathcal{C}$ file is shown in Figure 2. Of the many sorts of fluents mentioned in Figure 2, only inertial fluents are used in our formalization of the Yale Shooting domain. The syntax of constants and variables follows that of Prolog.[4]

The effect of the `include` declaration is the same as textually replacing the declaration by the contents of the included file.

The `sorts` declaration in Figure 2 declares three top-level sorts — `action`, `fluent`, and `time`. It declares that `step` is a subsort of `time`. (In general, `step` will be the set of all times except the last. These are the times at which actions can be performed.) It also declares (i) that `inertialFalseFluent` and `inertialTrueFluent` are subsorts of `fluent`, while `inertialFluent` is a subsort of each of these, and (ii) that `defaultFalseFluent` and `defaultTrueFluent` are subsorts of `fluent`, while `exogenousFluent` is a subsort of each of these. In `yale2.c` one additional sort, `gun`, is declared.

[3]The external database is available only when using SICStus Prolog. It is created only when the number of ground rules is large (¿ 200,000).

[4]Actually, it is legal to use a Prolog atom whose name begins with a lowercase letter as a variable. However, doing so is probably unwise, since it tends to make variables and non-variables (which must begin with a lowercase letter) difficult to distinguish.

```
% File: 'yale2.c'
:- include 'C'.

:- sorts gun.

:- variables
 G :: gun.

:- constants
 g1, g2                :: gun;
 load(gun), shoot(gun) :: action;
 alive, loaded(gun)    :: inertialFluent.

:- show all.

load(G) causes loaded(G).
shoot(G) causes -alive if loaded(G).
shoot(G) causes -loaded(G).

nonexecutable _A && _A1 if _A@<_A1.
```

Figure 1: The Yale Shooting Domain with Two Guns in $\mathcal{C}$

The **variables** declarations in Figure 2 and in **yale2.c** declare the needed variables and their sorts. The **constants** declarations declare the sorts of object, function, and predicates constants. Properly speaking, predicates do not have a result sort. However, the reserved word **atomicFormula** is written in place of a sort name.

Notice that in **yale2.c** the fluent **alive** is declared to be of sort **inertialFluent**. This declaration has the same effect as declaring **alive** to be a **fluent** (*simpliciter*) and then adding the $\mathcal{C}$ proposition

inertial alive, − alive.

Using the more specific declaration eliminates the need for the corresponding $\mathcal{C}$ proposition, and thus is more concise.[5]

The **show** declaration in **yale2.c** says that in displaying models **all** fluent literals should be shown. An alternative would be to declare instead

```
:- show positive.
```

which would cause only the positive fluent literals in a model to be shown, the negative ones being known by their absence. The default setting is to show only the positive fluent literals.

[5]Similar relationships hold for other subsorts of fluent, as is shown in the table below, where each sort declaration is paired with a corresponding $\mathcal{C}$ proposition.

```
: - constants
  f1 ::  inertialTrueFluent;                    inertial f1.
  f2 ::  inertialFalseFluent;                   inertial -f2.
  f3 ::  inertialFluent;                        inertial f3, -f3.
  f4 ::  defaultTrueFluent;                     default f4.
  f5 ::  defaultFalseFluent;                    default -f5.
  f6 ::  exogenousFluent;                       exogenous f6.
```

```
%% Standard sort declarations and associated causal laws in file 'C'.
:- sorts
 action;
 fluent
   >> ( ( inertialFalseFluent;
                 inertialTrueFluent    ) >> inertialFluent ;
        ( defaultFalseFluent;
                 defaultTrueFluent     ) >> exogenousFluent
      );
 time >> step.

:- variables
 _N,_N1,_N2 :: step;
 _A,_A1,_A2 :: action;
 _T,_T1,_T2 :: time;
 _F,_F1,_F2 :: fluent;
 _DF :: defaultFalseFluent;
 _DT :: defaultTrueFluent;
 _IF :: inertialFalseFluent;
 _IT :: inertialTrueFluent.

:- constants
 0    :: step;
 0, 1 :: time;
 o(action,step), h(fluent,time) :: atomicFormula.

h(_F,0)  =>  h(_F,0).
-h(_F,0) => -h(_F,0).

h(_IT,0) &&   h(_IT,1) =>  h(_IT,1).
-h(_IF,0) && -h(_IF,1) => -h(_IF,1).

h(_DT,1)  =>  h(_DT,1).
-h(_DF,1) => -h(_DF,1).

o(_A,0)  =>  o(_A,0).
-o(_A,0) => -o(_A,0).
```

Figure 2: Declarations from the Standard $\mathcal{C}$ File

It is also possible to specify exactly the fluent literals that one wishes to show. The syntax for this is explained in Section 5.

Here is an annotated example session. We begin by loading the file **yale2.c**.[6]

```
| ?- loadf('yale2.c').
% loading file /v/hank/v73/mccain/kk/yale2.c
% loading file /v/hank/v73/mccain/kk/C
% loading file /v/hank/v73/mccain/kk/yale2.p
% 10 atoms, 32 rules, 21 clauses, 90 msec.
yes
```

The file is loaded by calling `loadf/1` passing the file name as the argument. The language of the theory contains 10 ground atoms, and the generated ground theory — let us call it D — consists of 32 rules. The theory D is definite. Accordingly, the translation into propositional logic is carried out by the method of literal completion — let us call the resulting theory $lcomp(D)$.[7] In clausal form, $lcomp(D)$ reduces to 21 clauses after eliminating tautologies. The compilation process took 90 milliseconds.

```
| ?- plan.
enter facts (then ctrl-d)
|: 0: alive.
|: 0: (/\G: -loaded(G)).
|:
enter goals (then ctrl-d)
|: 2: -alive.
|:
calling rel_sat...

0.  alive -loaded(g1) -loaded(g2)

ACTIONS: load(g1)

1.  alive  loaded(g1) -loaded(g2)

ACTIONS: shoot(g1)

2. -alive -loaded(g1) -loaded(g2)

Elapsed Time (cpu sec): 0

Verify plan? (y/n) y
verifying..
The plan is verified!
yes
```

The procedure `plan/0` prompts for sets of time-labeled fluent formulas called *Facts* and *Goals*, and then seeks to determine whether or not there is a satisfying interpretation for

$$lcomp(D) \cup Facts \cup Goals. \qquad (1)$$

[6]Loading a file with a name that ends with `'.c'` (also, `'.t'`) causes CCALC to switch to transition mode. On the other hand, loading a `'.h'` file causes CCALC to switch to history mode.

[7]The method of literal completion was introduced in [McCain and Turner, 1997] and extended to the case with variables in [Lifschitz, 1997].

In the preceding problem, the *Facts* state that in the initial state the turkey is alive and both guns are unloaded. In the second mentioned fact, the formula following `0:` is an abbreviation for `(-loaded(g1) && -loaded(g2))`. The *Goals* (of which there is one) state that the turkey is dead at time 2. A satisfying interpretation is found and displayed, the plan is seen by examining the lines labeled `ACTIONS`. The plan in this case calls for doing exactly the following actions — loading gun `g1` at time 0 and then shooting it at time 1. The sat solver reports a time of 0 seconds to find this interpretation.

Finding a satisfying interpretation for (1) shows that there exists a causally possible world history in which the plan is executed (in the initial state of the interpretation) and the goal is achieved. It does not, however, guarantee that the plan is *valid*, i.e., that the plan will necessarily achieve the goal if the plan is executed in the initial state of the interpretation, or that it can be so executed. Furthermore, finding a satisfying interpretation does not guarantee that the initial state is completely specified by the given facts.

Consequently, after a satisfying interpretation — let us call it I — is found and displayed, the user is asked whether he wishes the Causal Calculator to attempt to verify the plan (i.e., the history of actions) contained in I. If he answers `'y'`, the Causal Calculator does the following two things.

1. It attempts to show that the causal theory and the given facts entail the initial state of I.
2. It attempts to show that the plan is *deterministic*, i.e., that for every time T ($0 \leq T < n$, where n is the largest integer in the range of times), the complete history up to and including time T, together with the actions performed and not performed at time T (as specified by the plan) entail the state of the world at time $T + 1$.

It is shown in [McCain and Turner, 1997] that every deterministic, causally possible plan is valid.[8]

Entering the facts and goal by hand can become tedious. If one knows in advance problems that one intends to pose, these problems can be described in either the input file or in separate files to be included. A separate problem file may be included in an input file by using the `include` command. It can also be included in the runtime environment after the input file has been loaded by using the `include` procedure at the Prolog top-level. It is necessary to use `include/1` rather than `loadf/1`, because `loadf/1` reinitializes the knowledge base.

We first include the problem file `yale2.p`, which is shown in Figure 3. The problem file `yale2.p` contains four planning problems, labeled 0–3, and one query problem, label 0. The labels in the first planning problem and the query problem are left out and assume the value 0 by default. A label may be any Prolog atom or integer. The order of the fields is unimportant. However, fields labeled `facts` and `goals` are required.

We pose predefined planning problems by calling `plan/1`, passing as an argument the label of the problem that we wish to pose. This is illustrated below.

```
| ?- include 'yale2.p'.
% loading file /v/hank/v38/mccain/cc/yale2.p

yes
```

[8]A simpler procedure for checking determinism — checking the determinism of a path in a transition system — is possible for domain descriptions written in $\mathcal{C}$. The method described here is required, however, for the more general class of theories specifiable in the primitive language of the Causal Calculator.

```
% File: yale2.p

:- plan
facts ::
 0: alive,
 0: (/\G: -loaded(G));
goals  ::
 2: -alive.

:- plan
label ::
 1;
facts ::
 0: -alive,
 0: loaded(g1);
goal  ::
 1: alive.

:- plan
label ::
 2;
facts ::
 0: alive,
 0: /\G:(-loaded(G));
goal  ::
 1..100: -alive.

:- query
facts ::
 0: (/\G: -o(shoot(G))),
 1: -alive;
goals ::
 0: -alive.

:- plan
label :: 3;
facts ::
 h(alive,0),
 (/\G: -h(loaded(G),0));
goals ::
 -h(alive,2).
```

Figure 3: Problems for the Yale Shooting Domain

```
| ?- plan(0).
calling rel_sat...

0.  alive -loaded(g1) -loaded(g2)

ACTIONS: load(g1)

1.  alive  loaded(g1) -loaded(g2)

ACTIONS: shoot(g1)

2. -alive -loaded(g1) -loaded(g2)

Elapsed Time (cpu sec): 0

Verify plan? (y/n) y
verifying..
The plan is verified!
yes
| ?- plan(1).
calling rel_sat...
Elapsed Time (cpu sec): 0.33
no
| ?- plan(2).

calling sato...
  run time (seconds)                 0.00
calling sato...
  run time (seconds)                 0.00

0.  alive -loaded(g1) -loaded(g2)

ACTIONS: load(g2)

1.  alive  loaded(g2) -loaded(g1)

ACTIONS: shoot(g2)

2. -alive -loaded(g1) -loaded(g2)

Verify plan? (y/n) y
verifying..
The plan is verified!
yes
```

The first planning problem posed (**plan(0)**) is the same one that we previously entered by hand.[9] In the second problem (**plan(1)**), we seek a plan to bring the turkey back to life. The answer **no** following the time reported in **rel_sat** shows that this is impossible. In the third problem (**plan(2)**), we pose a problem that is a variant of our first one. What is different

[9]The predicate plan/1 is defined as a prefix operator. Hence, it is possible to leave off the parentheses in the call, writing ?- plan 0.

in this case, is that we do not specify a single time at the which the goal is to be achieved, but instead specify a range of times `1..100`. Given this specification, CCALC first tries to find a plan that achieves the goal at time 1 (in histories with times in the range `0..1`). If this attempt fails, CCALC goes on to try to find a plan that achieves the goal at time 2 (in histories with times `0..2`), and so on, until the goal is achieved or the last element in the range of times is tried. There is no limit on the size of the specified range of times, but certain other restrictions apply (see Section 5.8).

Figure 3 contains a single query problem, with the default label 0. It is solved by calling the procedure `query/1`, passing the query label as an argument.

```
| ?- query(0).

calling sato...
  run time (seconds)                    0.00
yes
```

The procedure `query/1` seeks to determine whether the causal theory D and the *Facts* entail the conjunction of *Goals*. It does this by trying to determine whether there is a satisfying interpretation for $lcomp(D)$, *Facts*, and the negation of the conjunction of *Goals*. The conjunction of *Goals* is entailed just in case there is no such satisfying interpretation. The procedure `query/0` is identical to `query/1` except that, like `plan/0`, it prompts the user to enter the facts and goal from the terminal.

The time-labeled formulas that appear in plan and query problems are abbreviations for formulas that use the predicate `h/2` from the "query language" of the Causal Calculator. The expansion of time-labeled formulas can be seen in the following rewrite rule, which is defined in the standard $\mathcal{C}$ file.

```
:- macros
        #1 : #2 -> h(#2,#1)
                where integer(#1).
```

According to this rule, for example, the time-labeled formula `0: alive` expands into the formula `h(alive,0)`. This expansion occurs only when the term preceding the `':'` is an integer constant. Thus, the time-sequence labeled formula `1..100: -alive` is not expanded. Planning problem 3 of Figure 3 shows how problem 0 would be specified directly in the CCALC query language, without using the `':'` macro symbol. In this instance, there is no reason not to use the `':'` macro symbol, but in other cases there is. This is illustrated by the following query, which seeks to determine whether it follows from the fact that the turkey was alive at time 0 and dead at time 2 that at some time some gun was shot. Because the query involves a disjunctive condition over times, it is not possible to express it in the syntax of time labeled formulas.

In addition to the predicate `h/2`, the CCALC query language contains a second predicate `o/2`, which is read as "occurs". This is used in the following query. The sorts of the arguments to `h/2` and `o/2`, as well as the sort of the variable `_N`, are shown are shown in Figure 2.

```
| ?- query.
enter facts (then ctrl-d)
|: h(alive,0) && -h(alive,2).
|:
enter goals (then ctrl-d)
```

```
|: \/_N: \/G: o(shoot(G),_N).
|:
calling sato...
  run time (seconds)                0.00
yes
```

3 The Stuffy Room Domain

The Stuffy Room domain from [Ginsberg and Smith, 1988] is formalized in Figure 4. A planning problem is described in the second included file of Figure 5.

There are two aspects of the formalization in Figure 4 that deserve comment. First, notice that the fluents — `blocked(d1)`, `blocked(d2)`, and `stuffy` — are declared to be exogenous. These fluents are defined by `always` propositions, ultimately in terms of fluents of the form `on(O,L)`. The second thing to observe is that fluents of the latter form are not declared to be of sort `inertialFluent` (as were all fluents in our formalization of the Yale Shooting domain), but rather of sort `inertialTrueFluent`. Intuitively, this means that the truth of fluents of the form `on(O,L)` may be explainable by inertia, but their falsity cannot be. This weaker formulation of inertia suffices because the schema

```
caused -on(O,L1) if on(O,L) && -(L=L1).
```

provides explanations of a different kind for the falsity of fluents of the form `on(O,L)`. According to this schema, the absence of an object from a location is always explained by its presence at a different location. Arguably, this makes better sense than an explanation in terms of inertia. To see that this is so, imagine that object `o1` is moved from location `d2` to the `floor`, and consider the fact that afterwards `o1` is still not at location `d1`. Can the latter fact be explained by inertia? Intuitively, it cannot be, since `d1` was just moved.[10]

```
| ?- loadf('stuffy2.c').
% loading file /v/hank/v73/mccain/kk/stuffy2.c
% loading file /v/hank/v73/mccain/kk/C
% loading file /v/hank/v73/mccain/kk/stuffy2.p
% 24 atoms, 123 rules, 93 clauses, 170 msec.
yes
| ?- set(solver,sato).

yes
| ?- loadf('stuffy2.c').
% loading file /v/hank/v73/mccain/kk/stuffy2.c
% loading file /v/hank/v73/mccain/kk/C
% loading file /v/hank/v73/mccain/kk/stuffy2.p
% 24 atoms, 123 rules, 93 clauses, 180 msec.
yes
| ?- plan(0).
```

[10]This argument assumes that by declaring a fluent to be inertial we intend to describe a real property of the domain — at least, as we choose to model it. We are not merely describing a representation convention, since in that case the question of explanation would not arise. See [McCain and Turner, 1997, footnote 6], for another argument for the claim that inertial fluents do not necessarily come in positive and negative pairs.

```
% File: 'stuffy2.c'
:- include 'C'.

:- sorts location >> duct; object.

:- variables
 D      :: duct;
 L, L1 :: location;
 O, O1 :: object.

:- constants
 d1, d2                  :: duct;
 floor                   :: location;
 o1, o2                  :: object;
 on(object,location)     :: inertialTrueFluent;
 blocked(duct), stuffy   :: exogenousFluent;
 move(object,location)   :: action.

:- include 'stuffy2.p'.

move(O,L) causes on(O,L).
nonexecutable move(O,L) if on(O,L).

caused -on(O,L1) if on(O,L) && -(L=L1).

always \/L: on(O,L).
always on(O,D) && on(O1,D) ->> O=O1.

always blocked(D) <-> \/O:on(O,D).
always stuffy <-> blocked(d1) && blocked(d2).

nonexecutable _A && _A1 if _A@<_A1.
```

Figure 4: The Stuffy Room Domain in $\mathcal{C}$

```
% File: 'stuffy2.p'

:- plan
facts ::
 0: on(o1,d1),
 0: on(o2,d2);
goal  ::
 1..100: (on(o1,d2) && on(o2,d1)).
```

Figure 5: A Planning Problem for the Stuffy Room Domain

```
calling sato...
  run time (seconds)                 0.00
calling sato...
  run time (seconds)                 0.00
calling sato...
  run time (seconds)                 0.00

0.  stuffy  blocked(d1)  blocked(d2)  on(o1,d1)  on(o2,d2)

ACTIONS: move(o2,floor)

1.  blocked(d1)  on(o1,d1)  on(o2,floor)

ACTIONS: move(o1,d2)

2.  blocked(d2)  on(o1,d2)  on(o2,floor)

ACTIONS: move(o2,d1)

3.  stuffy  blocked(d1)  blocked(d2)  on(o1,d2)  on(o2,d1)

Verify plan? (y/n) y
verifying...
The plan is verified!
yes
```

In the preceding interaction, we choose `sato` as the satisfiability checker and then attempt to find a plan that will leave each of the objects in the other object's initial location. Sato finds that one possible plan is to carry out the following sequence of actions: `move(o2,floor)`; `move(o1,d2)`; `move(o2,d1)`.

4 The Monkey and Bananas Domain

A simple version of the Monkey and Bananas domain is formalized in Figure 6. The usual planning problem associated with this domain is defined in the included file `monkey.p` of Figure 7.

```
| ?- loadf('monkey.c').
% loading file /v/hank/v73/mccain/kk/monkey.c
% loading file /v/hank/v73/mccain/kk/C
% loading file /v/hank/v73/mccain/kk/monkey.p
% 30 atoms, 188 rules, 162 clauses, 230 msec.
yes
| ?- plan(0).

calling sato...
  run time (seconds)                 0.01
calling sato...
  run time (seconds)                 0.01
```

```
% File: 'monkey.c'
:- include 'C'.

:- sorts object; location.
:- variables
 O          :: object;
 L, L1, L2 :: location.

:- constants
 monkey, box, banana   :: object;
 l1, l2, l3            :: location;
 at(object,location)   :: inertialTrueFluent;
 onBox, hasBanana      :: inertialFluent;
 walk(location), pushBox(location), climbBox, graspBanana :: action.

 :- include 'monkey.p'.

walk(L) causes at(monkey,L).
nonexecutable walk(L) if at(monkey,L).
nonexecutable walk(L) if onBox.

pushBox(L) causes at(monkey,L).
pushBox(L) causes at(box,L).
nonexecutable pushBox(L) if at(monkey,L).
nonexecutable pushBox(L) if onBox.
nonexecutable pushBox(L) if at(monkey,L1) && at(box,L2) && -(L1=L2).

climbBox causes onBox.
nonexecutable climbBox if onBox.
nonexecutable climbBox if at(monkey,L) && at(box,L1) && -(L=L1).

graspBanana causes hasBanana.
nonexecutable graspBanana if -onBox.
nonexecutable graspBanana if at(monkey,L) && at(banana,L1) && -(L=L1).

caused -at(O,L1) if at(O,L) && -(L=L1).
caused at(banana,L) if at(monkey,L) && hasBanana.

always \/L:at(O,L).
always at(monkey,L) && onBox ->> at(box,L).

nonexecutable _A && _A1 if _A@<_A1.
```

Figure 6: The Monkey and Bananas Domain in $\mathcal{C}$

```
%File: 'monkey.p'

:- plan
facts ::
 0: at(monkey,l1),
 0: at(banana,l2),
 0: at(box,l3);
goals  ::
 1..100: hasBanana.
```

Figure 7: A Planning Problem for the Monkey and Bananas Domain

```
calling sato...
  run time (seconds)                 0.02
calling sato...
  run time (seconds)                 0.01

0.  at(banana,l2)  at(box,l3)  at(monkey,l1)

ACTIONS: walk(l3)

1.  at(banana,l2)  at(box,l3)  at(monkey,l3)

ACTIONS: pushBox(l2)

2.  at(banana,l2)  at(box,l2)  at(monkey,l2)

ACTIONS: climbBox

3.  onBox  at(banana,l2)  at(box,l2)  at(monkey,l2)

ACTIONS: graspBanana

4.  hasBanana  onBox  at(banana,l2)  at(box,l2)  at(monkey,l2)

Verify plan? (y/n) y
verifying....
The plan is verified!
yes
```

5 The Input File Syntax

An input file consists of declarations, comments, and propositions (or schemas) in $\mathcal{C}$. Declarations and propositions may appear in any order, except that macros and sorts must be defined before they are used. As in Prolog, the text appearing on a line following a `%` is taken as a comment, as is the text appearing on one or more lines between `/*` and `*/`.

Declarations are written as Prolog commands. They can be divided into two groups. The first group includes the following.

```
macros, include, sorts, variables, constants, tests
```

Each of these declarations takes a single argument which is a tuple of one or more elements. The elements of such a tuple are separated by semicolons. Multiple occurrences of one of the declarations in this group are equivalent to single occurrences in which the arguments are combined to form a single tuple (whose components are separated by semicolons). For example,

```
:- variables
 A :: action.
:- variables
 F :: fluent.
```

is equivalent to

```
:- variables
 A :: action;
 F :: fluent.
```

There are three other declaration commands that do not obey the abovementioned rule. They comprise the second group.

```
show, query, plan
```

Any command besides these (e.g., `:- compile(<file>)`) is handled by Prolog.

5.1 Macros

Macros are rewrite rules, which are specified using the `macros` declaration. The syntax is as follows.[11]

```
MacrosDeclaration ::= :- macros MacroTuple.
MacroTuple ::= Macro | ( MacroTuple ; MacroTuple )
Macro ::= Term -> Term | Term -> Term where Call
Term is any Prolog term.  Call is a Prolog call, except that it uses
macro variables in place of Prolog variables.
```

Macro variables, which are notated by `#1`, `#2`, `#3`, ... , may occur in the left and right `Term`s of a rewrite rule (Macro). However, every variable that occurs in the right `Term` should also either occur in the left or be bound to a ground term when Prolog executes Call. (Macro variables are replaced by Prolog variables during preprocessing.)

A term τ is said to be *rewritable* if there is a rewrite rule (α `->` β) and a substitution σ (mapping macro variables to terms) such that $\alpha\sigma = \tau$. In such a context, by *rewriting* τ we mean replacing it by $\beta\sigma$.[12]

[11] See Appendix A for an explanation of the grammar notation used. The parantheses in the second production below (and in later fragments of the grammar) are normally optional due to the precedences and associativity of operators. The precedences and associativity of all operators are given in Section 6.

[12] The notion of a rewritable term msut be modified to take account of rewrite rules that contain Prolog calls. Briefly, a term is said to be rewritable only if the call succeeds. Any bindings generated by executing the call are taken to extend σ.

Whenever the Causal Calculator reads in a term — whether from a file or the terminal — it examines the term for rewritable subterms, always rewriting the leftmost, outermost rewritable subterm next. Rewriting continues until no rewritable subterm remains.

The following macro declarations show the expansions (in transition mode) of dynamic and static laws in $\mathcal{C}$ into the language of causal theories.

```
:- macros
        caused #1 if #2 after #3
                -> (h(#3,0) && h(#2,1) => h(#1,1)) ;
        caused #1 if #2
                -> ( (h(#2,0) => h(#1,0));(h(#2,1) => h(#1,1)) ).
```

Other examples are shown in Appendix B (and D).

5.2 Include

The `include` declaration is used to include files during loading. The syntax is as follows.

```
IncludeDeclaration ::= :- include PathnameTuple.
PathnameTuple ::= Pathname | ( PathnameTuple ; PathnameTuple )
A Pathname is the name of a file, written as a Prolog atom.
```

The effect of `include` is the same as if the contents of the included files replaced the declaration itself.

The `include` command is usually included in an input file, but it can also be used at the Prolog top-level, *after* loading an input file. This may be useful for loading problem files, or files containing new macro declarations, variable declarations, or tests. Any of these may subsequently be used in posing planning problems. By contrast, since the translation to propositional logic is not modular, it makes no sense to try to add new sorts, constants, or propositions to a domain description after it is loaded. Rather the input file must be modified and the entire theory reloaded by using `loadf`.

5.3 Sorts

The `sorts` declaration is used to declare the sorts of the language of the domain description. The expression `a >> b` expresses that `a` and `b` are sorts and that `b` is a subsort of `a` — so all ground terms of sort `b` are ground terms of sort `a`. The semicolon may be read as "and". The syntax is as follows.

```
SortsDeclaration ::= :- sorts SortSpecTuple.
SortSpecTuple ::= SortSpec | ( SortSpecTuple ; SortSpecTuple )
SortSpec ::= SortName | ( SortNameTuple >> SortSpecTuple )
SortNameTuple ::= SortName | ( SortNameTuple ; SortNameTuple )
A SortName is any Prolog atom.
```

The grammar allows the complex sort specifications shown on the lefthand sides of the following rewrite rules, where `N1` and `N2` are `SortNameTuple` and `S1` and `S2` are `SortSpecTuple`.

```
N1 >> (N2 >> S1)   ->  (N1 >> N2) ; (N2 >> S1)
N1 >> (S1 ; S2)    ->  (N1 >> S1) ; (N1 >> S2)
(N1 ; N2) >> S1    ->  (N1 >> S1) ; (N2 >> S1)
```

The meanings of the complex forms are fully determined by these rewrite rules.

There are two restrictions on subsort specifications. First, a sort cannot be defined—either directly or indirectly—to be a subsort of itself. Secondly, a sort can be defined to be a subsort of two sorts only if the latter sorts are already subsorts of a common sort. Violations of the both restrictions are caught and reported by the Causal Calculator.

5.4 Variables

The `variables` declaration is used to declare variables for use in a domain description and in specifying problems. All variables are global and sorted. The syntax is as follows.

```
VariablesDeclaration ::= :- variables VariablesSpecTuple.
VariablesSpecTuple ::= VariablesSpec
                       | ( VariablesSpecTuple ; VariablesSpecTuple )
VariablesSpec ::= ( VariablesList :: SortName )
VariablesList ::= Variable | ( Variable , VariablesList )
A Variable is any Prolog atom.
```

Although one may choose to use any Prolog atom as a variable, since non-variable atoms begin with a lowercase letter, it is recommended that one adhere to the Prolog syntax for variables – choosing variable names that begin with a capital letter or an underscore.

5.5 Constants

The `constants` declaration is used to declare the non-logical constants of the language of the domain description. The syntax is as follows.

```
ConstantsDeclaration ::= :- constants ConstantsSpecTuple.
ConstantsSpecTuple ::= ConstantsSpec
                       | ( ConstantsSpecTuple ; ConstantsSpecTuple )
ConstantsSpec ::= ( ConstantSchemaList :: SortName )
                  | ( ( Integer .. Integer ) :: SortName )
ConstantSchemaList ::= ConstantSchema
                       | ( ConstantSchema , ConstantSchemaList )
ConstantSchema ::= Constant | Atom(SortName,...,SortName)
Constant and Atom are as in Prolog.  (An Integer is a Constant.)
```

The expression `M..N`, for integers `M` and `N`, is a abbreviation for the sequence of integers from `M` to `N`, inclusive.

5.6 Tests

The `tests` declaration is used to declare so-called 'test' predicates. The syntax is as follows.

```
TestsDeclaration ::= :- tests TestsSpecTuple.
TestsSpecTuple ::= TestsSpec | ( TestsSpecTuple ; TestsSpecTuple )
TestsSpec ::= Functor/Arity
Functor is a Prolog atom and Arity is an natural number.
```

Any built-in or user-defined predicate in Prolog can be declared to be a test. The following are predeclared in the Causal Calculator.

```
:- tests true/0; false/0; next/2; =/2; </2; >/2; <=/2; >=/2;
         @</2; @>/2; @<=/2; @>=/2; is/2.
```

With the exception of **next/2**, each of the built-in tests is predefined in Prolog. **next/2** is defined in the Causal Calculator as follows.

```
next(T,T1) :- T1 is T+1.
```

Tests are evaluated after they are instantiated. In order to explain how this is done, consider the following proposition, where K, M, and N are variables ranging over the domain 0..5.

```
earn(K) causes has(N) if has(M) && N is M+K.
```

After K, M, and N are instantiated to specific integers in the domain 0..5, N is M+K is evaluated, calling the Prolog procedure is/2. If the call succeeds, it is dropped from the conjunctive antecedent. If it fails, then no instances of the proposition (or, more precisely, no instances of the corresponding schema in the language of causal theories) with the current substitutions are generated. In the general case, a test call is replaced by either **true** or **false**, depending on whether the call succeeds or fails.

It is generally convenient to include the Prolog file containing the definitions of user-defined test predicates by including a **compile** command in the Causal Calculator input file. For example, if we had the need for tests beyond those that are built in, we might define them by Prolog procedures in a file named **tests.pl** and add the following line in the file **yale2.c**.

```
:- compile('tests.pl').
```

Atomic formulas containing test predicates are allowed everywhere that ordinary atomic formulas are allowed.

5.7 Show

The **show** declaration is used to specify which fluents should be written out when displaying a model. The syntax is as follows.

```
ShowDeclaration ::= :- show ShowSpecTuple.
ShowSpecTuple ::= ShowSpec | ( ShowSpecTuple ; ShowSpecTuple )
ShowSpec ::= all | positive | negative | none | [-]Functor/Arity
Functor is a Prolog atom and Arity is an natural number.
```

A `show` declaration does not augment a previous show declaration (if any) but rather replaces it.

The arguments `all`, `positive`, `negative`, and `none` mean to show all, only the positive, only the negative, and no fluent literals, respectively. Arguments of the form `[-]Functor/Arity` specify specific fluent literals to show. As an example, replacing the show declaration in Figure 1 by

```
:- show alive/0; loaded/1; -loaded/1.
```

would cause both positive and negative fluent literals for `loaded` but only the positive fluent literals for `alive` to be shown.

5.8 Plan and Query

The `plan` and `query` declarations are used to specify query and planning problems. The syntax is as follows.

```
PlanDeclaration ::= :- plan PlanFieldTuple.
PlanFieldTuple ::= (PlanFieldTuple ; PlanFieldTuple) | PlanField
PlanField ::= label :: Constant | time :: Integer
            | { facts | fact } :: QLFormulaTuple
            | { goals | goal } :: { QLFormulaTuple | TimeSeq: FluentFormula }

QueryDeclaration ::= :-  query QueryFieldTuple.
QueryFieldTuple ::= (QueryFieldTuple ; QueryFieldTuple) | QueryField
QueryField ::= label :: Constant | time :: Integer
            | { facts | fact } :: QLFormulaTuple
            | { goals | goal } :: QLFormulaTuple

QLFormulaTuple ::= QLFormula | ( QLFormulaTuple; QLFormulaTuple )
TimeSeq ::= Integer..Integer
QLFormula and Fluentformula are described in the next section.
```

The `facts` and `goals` fields are required. The `label` field is optional and defaults to the label `0`. The `time` field is also optional. It defaults to the maximum time mentioned in any of the `TimedFormula`'s. If a field name appears more than once, the second and later occurrences are ignored.

A `TimeSeq` $t_1..t_n$ must satisfy the following restrictions: (1) t_1 and t_n are natural numbers, (2) $t_n \geq 1$, (3) $t_1 \leq t_n$, (4) t_1 must be a least as large as the maximum time label of any given fact. Notice that when a `TimeSeq` is used, only one goal formula may be given.

5.9 Formulas

The reader is assumed to know the syntax of propositions in $\mathcal{C}$, including abbreviations that include the keywords `causes`, `nonexecutable`, `always`, and so on. Any doubts may be resolved by examining the macro declarations in Appendix B (or D). These declarations contain one extension to the $\mathcal{C}$ language that is not described in [Gelfond and Lifschitz, 1998]. This is the functor `o/1`. Given an action symbol a, the term `o(a)` is a fluent symbol which says that the action a occurs. This is illustrated by the following query wrt the domain formalized in Figure 1.

```
| ?- query.
enter facts (then ctrl-d)
|: 0: (loaded(g1) && o(shoot(g1))).
|:
enter goals (then ctrl-d)
|: 1: -alive.
|:
calling rel_sat...
Elapsed Time (cpu sec): 0.32
yes
```

The word "formula" is used in $\mathcal{C}$ to refer to propositional combinations of action and fluent names. Here, except in certain special contexts, action names are required to be wrapped in `o/1`, as in the case of `shoot(g1)` above. The special contexts where `o/1` is not used to wrap action names can be seen by examining the macro definitions in the Appendix B. The macros corresponding to the keywords **`causes`**, **`nonexecutable`**, and **`may cause`** each wraps `o/1` itself around action expressions, so `o/1` should not be explicitly used in these contexts.

Since `o/1` transforms action names into fluent names, in place of **`Formula`**, we use the term **`FluentFormula`**. We use the latter term to refer to expressions that represent families of propositional combinations of formulas by means of schematic variables.

The syntax for **`FluentFormula`** is defined below. In the case of conjunction, several alternative symbols are allowed.

```
FluentFormula ::=  AtomicFluentFormula
             | ( FluentFormula <-> FluentFormula )
             | ( FluentFormula ->> FluentFormula )
             | ( FluentFormula ++ FluentFormula )
             | ( FluentFormula Conj FluentFormula )
             | ( - FluentFormula )
             | ( ( /\ Variable ) : FluentFormula )
             | ( ( \/ Variable ) : FluentFormula )

Conj ::= && | & | ,
```

An **`AtomicFluentFormula`** may be a schematic test atom, a schematic constant of sort fluent (perhaps formed from a schematic action constant and `o/1`), or a fluent variable.

Formulas of the last two listed forms

```
/\X: F
\/X: F
```

are called "multiple conjunctions" and "multiple disjunctions," respectively. They should be used only when the range of the metavariable X is non-empty. The Causal Calculator does not check for violations of this condition.

White space is sometimes forbidden or required by the Prolog reader. For example, in the following formula,

```
/\O: \/L: on(O,L)
```

white space is required after the first colon, but is forbidden between the characters `o(`. As in the case of declarations, many of the parentheses in formulas can be dropped due to operator precedences.

The syntax of `QLFormula` uses the predicates `h/2` and `o/2`.

```
QLFormula ::=  AtomicQLFormula
             | ( QLFormula <-> QLFormula )
             | ( QLFormula ->> QLFormula )
             | ( QLFormula ++ QLFormula )
             | ( QLFormula Conj QLFormula )
             | ( - QLFormula )
             | ( ( /\ Variable ) : QLFormula )
             | ( ( \/ Variable ) : QLFormula )

AtomicQLFormula ::= h(Fluent,Time) | o(Action,Time)
```

The relationship `FluentFormula` and `QLFormula` can be gleaned from the macro declarations involving `h/2`, `o/1`, and `o/2` in Appendix B (and D).

6 Operators and their Precedences

The following are the standard operators in Prolog. (This listing is taken from the SICStus Prolog manual.) The first argument to `op` defines an operator's precedence. The functor with the highest precedence is the principal functor in any unparenthesized expression. Lower precedences bind more tightly. The second argument indicates whether the operator is prefix (`fx`), infix (`xfx`, `xfy`, `yfx`), or postfix (`xf`). In the case of an infix operator, (`yfx`) is used to indicate whether the operator is "left associative," and (`xfy`) is used indicate that it is "right associative." The third argument is the operator or a list of operators.

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic,
                    multifile, block, meta_predicate,
                    parallel, sequential ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, is, =.., ==, \==, @<, @>, @=<, @>=,
                                  =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, #, /\, \/ ]).
:- op(  500,  fx, [ +, - ]).
:- op(  400, yfx, [ *, /, //, <<, >> ]).
:- op(  300, xfx, [ mod ]).
:- op(  200, xfy, [ ^ ]).
```

In addition to these standard Prolog operators, the Causal Calculator defines the following operators.

```
:- op(1150,fx,[include, macros, sorts, variables, constants, tests,
show, reshow, plan, query]).

% the causal operator in various guises
:- op(1150,xfx,[ =>, <=, <- ]).
:- op(1150,fx,[ <=, <- ]).

% logical operators
:- op(800,xfy,<->).          % if and only if
:- op(780,xfy,->>).          % only if
:- op(770,xfy,++).           % or
:- op(760,xfy,[ &&, & ]).    % and
:- op(300, fx,/\).           % multiple conjunctions
:- op(300, fx,\/).           % multiple disjunctions

% special operators
:- op(1070,xfx,[::]).        % used in declarations
:- op(1000,xfy,[x ]).        % this is reserved for later use
:- op(700,xfx,[.. ]).        % used in constants declaration
:- op(400,xfy,[>> ]).        % >> is redefined to be right associative.
:- op(200,fx,[#   ]).        % used in macro variables #1, #2, ...

% The following operators are defined in Prolog.
%:- op(1000,xfy,[(;)]).      % used in declarations
%:- op(1050,xfy,[ ->]).      % used in macro definitions
%:- op(1000,xfy,[',']).      % used in declarations
%:- op(550,xfy,[   :]).      % used with /\X and \/X
%:- op(500,fx,[    -]).      % the sign for logical negation
```

As a consequence of these operator declarations, many parentheses appearing in the grammar rules listed in Section A may be dropped. For example, the variable delcaration

```
:- variables
 ((G      :: gun);
 (((A, A1) :: action);
 (((T, T1) :: time)))).
```

can be written as

```
:- variables
 G      :: gun;
 A, A1 :: action;
 T, T1 :: time.
```

and the formula `((a ++ b) ->> c)` can be written as `a ++ b ->> c`, if desired.

7 User-Defined Parameters

The Causal Calculator contains a small set of user-defined parameters which affect its behavior. These parameters and their current values can be displayed as follows.

```
| ?- show_parameters.

    solver = rel_sat
    timed = true
    sorted = true
    mode = history
yes
```

The value of the parameter **solver** may be the name of either of the two satisfiability checkers, **rel_sat** and **sato**. It controls which of the sat solvers is used. The values of the parameters **timed** and **sorted** may be either **true** or **false**. The **timed** parameter controls whether or not the cpu time spent in satisfiability checking is displayed. The **sorted** parameter controls whether or not the input clause file passed to the satisfiability checker is sorted by length. Sorting often has a beneficial effect on runtimes. The initial values of the parameters are those shown above.

A parameter may be changed by using the procedure **set/2**. For instance, in order to manually switch to transition mode, one would enter the following, after which the parameters would have the values shown.

```
| ?- set(mode,transition).

yes
| ?- show_parameters.

    solver = rel_sat
    timed = true
    sorted = true
    mode = transition
yes
```

The procedure **reset_parameters/0** can be used to set all of the parameters back to their initial values.

The **mode** parameter is set to have the value **transition** when a file with the extension **'.c'** or **'.t'** is loaded. It is set to have the value **history** when a file with the extension **'.h'** is loaded. All parameters retain their last assigned values until they are explicitly or implicitly set, or CCALC is reconsulted or recompiled.

In addition to the procedure **show_parameters/0**, the following **show**-procedures are also provided: **show_atoms/0** **show_rules/0**, **show_schemas/0**, **show_clauses/0**, **show_queries/0**, and **show_plans/0**. These procedures display all expressions of the indicated types and thus are useful only when the numbers of existing expressions are relatively small. By contrast, the procedures **show_query/1**, **show_plan/1**, and **show_rules/1** allow one to display a selected subset of the existing expressions. The procedures **show_query/1** and **show_plan/1** display the single query or plan with the given argument label, while the procedure **show_rules/1** displays the subset of rules that contain a head literal that unifies with the given pattern. For example, having loaded the file **'yale.c'** of Figure 1, we obtain the following behavior.[13]

```
| ?- show_rules(h(loaded(_),0)).
```

[13] Note that the variable in the given pattern is not one of the sorted variables used in describing the domain. Rather, it is a Prolog variable.

```
h(loaded(g1),0) <- h(loaded(g1),0).
h(loaded(g2),0) <- h(loaded(g2),0).
yes
```

The syntax in which rules are displayed is one possible syntax for input rules in the primitive language of the Causal Calculator. The rules may be simpler in two ways than the input rules or schemas. First, test predicates are evaluated away during grounding. Secondly, rules containing complex formulas in the antecedents or consequents of input rules may be replaced by a set of simpler rules, each of which has a conjunction of literals in its antecedent and a disjunction of literals in its consequent.[14]

Acknowledgements

Thanks to Vladimir Lifschitz for many of the design ideas relating to the Causal Calculator, including the macro facility and its role in translating $\mathcal{C}$ domain descriptions. This work was partially supported by NSF grant IIS-9732744 and by the Texas Advanced Research Program under grant no. 003658-242.

References

[Bayardo and Schrag, 1997] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. of AAAI-97*, pages 203–208, 1997.

[Gelfond and Lifschitz, 1998] Michael Gelfond and Vladimir Lifschitz. Action Languages. Electronic Transactions on AI, Vol. 3, No. 16, 1998.

[Ginsberg and Smith, 1988] Mathew Ginsberg and Davide Smith. Reasoning about action I: a possible worlds approach. *Artificial Intelligence*, 35(2):165–195, 1988.

[Giunchiglia and Lifschitz, 1998] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: preliminary report. In *Proc. AAAI-98*, pages 623-630, 1998.

[Hanks and McDermott, 1987] Steve Hanks and Drew McDermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. of ECAI-92*, pages 359–379, 1992.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, 1996.

[Lifschitz, 1997] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.

[Plaisted and Greenbaum, 1986] B.A. Plaisted and S. Greenbaum. A structure preserving clause form translation. In the Journal of Symbolic Computation, pages 293–304, 1986.

[McCain and Turner, 1997] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. of AAAI-97*, pages 460–465, 1997.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability Planning with Causal Theories. In the *Proceedings of the Sixth International Conference on the Principles*

[14] The number of rules displayed by `loadf/1` is the number of rules after simplification.

of *Knowledge Representation and Reasoning (KR'98)*, pp. 212-223. (A preliminary version appeared in the papers for the AAAI-97 workshop on Robots, Softbots, Immobots: Theories of Action, Planning and Control.)

[McCain, 1999] Norman McCain. A Manual for the Causal Calculator. Unpublished draft, 1999.

[Zhang, 1997] Zhang, H. SATO: An Efficient Propositional Prover. In the Proceedings of the International Conference on Automated Deduction (CADE-97).

A Syntax

An input file consists of declarations and propositions (or schemas) in $\mathcal{C}$. No order of presentation is assumed, though macros and sorts must be defined before they can be used.

The syntax of declarations is defined in the partial EBNF grammar below. The grammar is called "partial" because at certain points it appeals to the syntax of Prolog terms. The following symbols belong to the EBNF notation.

`::=, |, [ ], { }`

The vertical bar expresses disjunction, the square brackets mean that what they enclose optional, and the braces are used for grouping. Nonterminals begin with a capital letter. Many of the parentheses in the following grammar can be dropped due to operator precedences. (The precedences of operators are defined in the next section.)

If Prolog allows a term to be written in infix, prefix, or postfix notation, the same notation may be used in place of the canonical notation used below. For instance, `X=Y` can be written in place of `=(X,Y)`.

```
MacrosDeclaration ::= :- macros MacroTuple.
MacroTuple ::= Macro | ( MacroTuple ; MacroTuple )
Macro ::= Term -> Term | Term -> Term where Call
Term is any Prolog term.  Call is a Prolog call, except that it uses
macro variables in place of Prolog variables.

IncludeDeclaration ::= :- include PathnameTuple.
PathnameTuple ::= Pathname | ( PathnameTuple ; PathnameTuple )
A Pathname is the name of a file, written as a Prolog atom.

SortsDeclaration ::= :- sorts SortSpecTuple.
SortSpecTuple ::= SortSpec | ( SortSpecTuple ; SortSpecTuple )
SortSpec ::= SortName | ( SortName >> SortSpecTuple )
SortNameTuple ::= SortName | ( SortNameTuple ; SortNameTuple )
A SortName is any Prolog atom.

VariablesDeclaration ::= :- variables VariablesSpecTuple.
VariablesSpecTuple ::= VariablesSpec
                       | ( VariablesSpecTuple ; VariablesSpecTuple )
VariablesSpec ::= ( VariablesList :: SortName )
VariablesList ::= Variable | ( Variable , VariablesList )
A Variable is any Prolog atom.
```

```
ConstantsDeclaration ::= :- constants ConstantsSpecTuple.
ConstantsSpecTuple ::= ConstantsSpec
                     | ( ConstantsSpecTuple ; ConstantsSpecTuple )
ConstantsSpec ::= ( ConstantSchemaList :: SortName )
                | ( ( Integer .. Integer ) :: SortName )
ConstantSchemaList ::= ConstantSchema
                     | ( ConstantSchema , ConstantSchemaList )
ConstantSchema ::= Constant | Atom(SortName,...,SortName)
Constant and Atom are as in Prolog.  (An Integer is a Constant.)

ShowDeclaration ::= :- show ShowSpecTuple.
ShowSpecTuple ::= ShowSpec | ( ShowSpecTuple ; ShowSpecTuple )
ShowSpec ::= all | positive | negative | none | [-]Functor/Arity
Functor is a Prolog atom and Arity is an natural number.

TestsDeclaration ::= :- tests TestsTuple.
TestSpecTuple ::= TestSpec | ( TestSpecTuple ; TestSpecTuple )
TestSpec ::= Functor/Arity
Functor is a Prolog atom and Arity is an natural number.

PlanDeclaration ::= :- plan PlanFieldTuple.
PlanFieldTuple ::= (PlanFieldTuple ; PlanFieldTuple) | PlanField
PlanField ::= label :: Constant | time :: Integer
            | { facts | fact } :: QLFormulaTuple
            | { goals | goal } :: { QLFormulaTuple | TimeSeq: FluentFormula }

QueryDeclaration ::= :-  query QueryFieldTuple.
QueryFieldTuple ::= (QueryFieldTuple ; QueryFieldTuple) | QueryField
QueryField ::= label :: Constant | time :: Integer
            | { facts | fact } :: QLFormulaTuple
            | { goals | goal } :: QLFormulaTuple

QLFormulaTuple ::= QLFormula | ( QLFormulaTuple; QLFormulaTuple )
TimeSeq ::= Integer..Integer
QLFormula and Fluentformula are described in the next section.

Formula ::=  AtomicFormula
            | ( Formula <-> Formula )
            | ( Formula ->> Formula )
            | ( Formula ++ Formula )
            | ( Formula Conj Formula )
            | ( - Formula )
            | ( ( /\ Variable ) : Formula )
            | ( ( \/ Variable ) : Formula )

Conj ::= && | & | ,
```

An **AtomicFormula** is either a test atom or (when using the $\mathcal{C}$ input language) a fluent symbol, possibly contructed from an action symbol and the functor `o/2` (see Section 5.9).

B The Standard $\mathcal{C}$ File: Transition Mode

The standard $\mathcal{C}$ file consists of three main sections: (i) operator declarations, (ii) macros for translating $\mathcal{C}$ into causal logic, and (iii) declarations of a number of standard sorts and causal laws.

```
% File: 'C' -- for use with C in transition mode; times 0 and 1.

%% Operators for the language {\cal C}.
:- op(1010,fx,caused).
:- op(1010,xfx,causes).
:- op(1010,xf,may).
:- op(1010,yfx,cause).
:- op(1020,xfx,if).
:- op(1030,xfx,after).
:- op(1010,fx,inertial).
:- op(1010,fx,exogenous).
:- op(1010,fx,nonexecutable).
:- op(1010,fx,default).
:- op(1010,fx,always).
:- op(1010,fx,never).
:- op(1010,fx,constant).

%% Macros translating {\cal C} propositions into causal laws.  An alternative
%% to the translation given below for static laws would be
%%        caused #1 if #2
%%                -> ( (h(#2,0) => h(#1,0));(h(#2,1) => h(#1,1)) ).
%% If every fluent has its value at time zero determined exogenously, the two
%% translations are equivalent.

:- macros
        caused #1 if #2 after #3
                -> (h(#3,0) && h(#2,1) => h(#1,1)) ;
        caused #1 if #2
                -> ( (h(#2,0) ->> h(#1,0)) ; (h(#2,1)  => h(#1,1)) );

        caused #1 after #3
                -> caused #1 if true after #3 ;
        caused #1
                -> caused #1 if true ;

        #1 causes #2 if #3
                -> caused #2 if true after #3 && o(#1) ;
        #1 causes #2
                -> caused #2 if true after o(#1) ;

        inertial #1,#2
                -> (inertial #1; inertial #2) ;

        inertial #1
                -> caused #1 if #1 after #1 ;
```

```
exogenous #1,#2
        -> (exogenous #1; exogenous #2) ;

exogenous #1
        -> ( caused #1 if #1
           ; caused -(#1) if -(#1)) ;

never #1
        -> caused false if #1 ;
always #1
        -> caused false if -(#1) ;

nonexecutable #1 if #2
        -> caused false after o(#1) && #2 ;
nonexecutable #1
        -> caused false after o(#1) ;

default #1 if #2
        -> caused #1 if #1 && #2 ;
default #1
        -> caused #1 if #1 ;

#1 may cause #2 if #3
        -> caused #2 if #2 after #3 && o(#1) ;
#1 may cause #2
        -> caused #2 if #2 after o(#1) ;

constant #1
        -> ( caused false if #1 after -(#1)
           ; caused false if -(#1) after #1 );

h(-(#1),#2)
        -> -h(#1,#2) ;
h((#1 && #2),#3)
        -> h(#1,#3)  && h(#2,#3) ;
h((#1  & #2),#3)
        -> h(#1,#3)   & h(#2,#3) ;
h((#1  , #2),#3)
        -> h(#1,#3)   , h(#2,#3) ;
h((#1 ++ #2),#3)
        -> h(#1,#3)  ++ h(#2,#3) ;
h((#1 ->> #2),#3)
        -> h(#1,#3) ->> h(#2,#3) ;
h((#1 <-> #2),#3)
        -> h(#1,#3) <-> h(#2,#3) ;
h((/\ #1 : #2),#3)
        -> /\ #1 : h(#2,#3) ;
h((\/ #1 : #2),#3)
        -> \/ #1 : h(#2,#3) ;
```

```
        o(-(#1),#2)
                -> -o(#1,#2) ;
        o((#1 && #2),#3)
                -> o(#1,#3)  && o(#2,#3) ;
        o((#1  & #2),#3)
                -> o(#1,#3)   & o(#2,#3) ;
        o((#1  , #2),#3)
                -> o(#1,#3)   , o(#2,#3) ;
        o((#1 ++ #2),#3)
                -> o(#1,#3)  ++ o(#2,#3) ;
        o((#1 ->> #2),#3)
                -> o(#1,#3) ->> o(#2,#3) ;
        o((#1 <-> #2),#3)
                -> o(#1,#3) <-> o(#2,#3) ;

        o((/\ #1 : #2),#3)
                -> /\ #1 : o(#2,#3) ;
        o((\/ #1 : #2),#3)
                -> \/ #1 : o(#2,#3) ;

        h(o(#1),#2)
                -> o(#1,#2).

%% A few auxilliary macros, some with Prolog 'where' clauses.
:- macros
        #1 : #2 -> h(#2,#1)
                where integer(#1) ;
        #1 - #2 -> #3
                where integer(#1), integer(#2), #3 is #1 - #2 ;
        #1 + #2 -> #3
                where integer(#1), integer(#2), #3 is #1 + #2.


%% Standard sort declarations and associated causal laws.
:- sorts
 action;
 fluent
   >> ( ( inertialFalseFluent;
                 inertialTrueFluent    ) >> inertialFluent ;
        ( defaultFalseFluent;
                 defaultTrueFluent     ) >> exogenousFluent
      );
 time >> step.

:- variables
 _N,_N1,_N2 :: step;
 _A,_A1,_A2 :: action;
 _T,_T1,_T2 :: time;
 _F,_F1,_F2 :: fluent;
 _DF :: defaultFalseFluent;
 _DT :: defaultTrueFluent;
```

```
 _IF :: inertialFalseFluent;
 _IT :: inertialTrueFluent.

:- constants
 0    :: step;
 0, 1 :: time;
 o(action,step), h(fluent,time) :: atomicFormula.

h(_F,0)  =>  h(_F,0).
-h(_F,0) => -h(_F,0).

h(_IT,0) &&   h(_IT,1) =>  h(_IT,1).
-h(_IF,0) && -h(_IF,1) => -h(_IF,1).

h(_DT,1)  =>  h(_DT,1).
-h(_DF,1) => -h(_DF,1).

o(_A,0)  =>  o(_A,0).
-o(_A,0) => -o(_A,0).
```

C History Mode

A domain description in the action language $\mathcal{C}$ describes a transition system, consisting of a set of states – a subset of the set of interpretations of the fluent symbols – and a transition relation that holds between a state s, an interpretation a of the action symbols, and a state s'. Two kinds of causal laws are expressible in $\mathcal{C}$: (i) `dynamic laws` that describe conditions for fluent values in s' in terms of fluent values in s and s', and the values of action symbols in a, and (ii) `static laws` that describe conditions in s' for fluent values in s' (and that also constrain the set of possible states).

The primitive language of the Causal Calculator is the language of causal theories with the predicates `h/2` and `o/2`. The predicates `h/2` and `o/2` are read as "holds" and "occurs," respectively. `h(f,0)` says that the fluent `f` holds at time 0. `o/2` says that the action `a` occurs at time 0.

Causal laws in $\mathcal{C}$ correspond to two special forms of causal laws in the language of causal theories. The correspondence is given by the macro rules from the standard $\mathcal{C}$ file shown below.

```
:- macros
        caused #1 if #2 after #3
                -> (h(#3,0) && h(#2,1) => h(#1,1)) ;
        caused #1 if #2
                -> ( (h(#2,0) ->> h(#1,0)) ; (h(#2,1)  => h(#1,1)) );
```

Notice that causal laws in $\mathcal{C}$ expand into expressions in the language of causal theories that mention only times 0 and 1.

In the language of causal theories, unlike the language $\mathcal{C}$, there are no built in assumptions about the temporal relations of conditions and effects. Casual laws may relate conditions at three or more times, rather than just one or two. Also, effects need not be said to immediately follow in time or hold simultaneously with their conditions. They may, for example, be delayed.[15]

Because of the greater expressiveness of the language of causal theories, its semantics is not defined by a transition system. Instead, the semantics is given in terms of the notion of a "causally explained interpretation" [McCain and Turner, 1997] of the underlying propositional language of the theory. If the underlying language contains explicit references to time – as is the case in the primitive language of the Causal Calculator – the interpretations of the language correspond to histories of the formalized domain, and the causally explained interpretations correspond to "causally possible" histories of the domain, according to the theory.

In transition mode, CCALC assumes that a causal theory involving times 0 and 1 is given. (The theory may be given in the language $\mathcal{C}$ or in the language of causal theories directly.) This theory is "shifted" to obtain a theory of histories over a range of times specified in a particular planning problem or query. In history mode, on the other hand, a causal theory about all times in a range of times (specified with the theory itself rather than with a particular problem) is obtained, not by shifting, but directly by input. In the primitive language of the Causal Calculator, facts are specified as conditions for facts being caused, regardless of time.

When using the $\mathcal{C}$ input language, there is no reason to use history mode. By using transition mode, one obtains faster compilation (i.e., grounding and conversion to clauses), since the causal theory involves only two times, 0 and 1. and one also obtains greater flexibility, since the length of histories is specified at a later stage and may be changed from one problem to the next.

Nevertheless, it is possible to use history mode even with the language $\mathcal{C}$. (This may be useful if one formalizes a domain partly in $\mathcal{C}$ and partly in the language of causal theories.) To do so, one must declare the macro symbol `maxtime` in the input file. `maxtime` defines the range of times `0..maxtime` that exist in histories of the domain. Besides these two declaration, it is only necessary to replace the standard $\mathcal{C}$ file `C` by its variant `C.h`, which is listed in Appendix D. `C.h` is identical to `C`, except that in `C.h` variables of types `time` (and `step`, a type which includes all times except the last) replace the constants 0 and 1 in `C`. A $\mathcal{C}$ file for the Yale shooting domain is shown in Figure 8.

The file `'yale2.h'` is loaded below. Notice the difference in the numbers of atoms, rules, and clauses generated here and in Section 2. The same three planning problems may be run as before. In the case of planning problem 2, which involves backtracking over a sequence of times, it is important to know that in history mode only the time of the goal is changed in different trials, not the range of times in possible histories. In history mode, the range of times in possible histories is specified in the input file and cannot be changed.

```
| ?- loadf('yale2.h').
% loading file /v/hank/v73/mccain/kk/yale2.h
% loading file /v/hank/v73/mccain/kk/C.h
% loading file /v/hank/v73/mccain/kk/yale2.p
% 17 atoms, 58 rules, 42 clauses, 170 msec.
yes
| ?- plan(0).

calling sato...
```

[15]Indeed, sufficient conditions for facts being caused may even hold later in time than the facts themselves. In the language of causal theories, no assumption about the relationship between time and causation is presupposed.

```
% File: 'yale2.h'

:- macros maxtime -> 2.
:- include 'C.h'.

:- sorts gun.
:- variables G :: gun.

:- constants
 g1, g2                :: gun;
 load(gun), shoot(gun) :: action;
 alive, loaded(gun)    :: inertialFluent.

:- include 'yale2.p'.
:- show all.

load(G) causes loaded(G).
shoot(G) causes -alive if loaded(G).
shoot(G) causes -loaded(G).

nonexecutable _A && _A1 if _A@<_A1.
```

Figure 8: The Yale Shooting Domain in History Mode

```
  run time (seconds)                 0.01

0.  alive -loaded(g1) -loaded(g2)

ACTIONS: load(g2)

1.  alive  loaded(g2) -loaded(g1)

ACTIONS: shoot(g2)

2. -alive -loaded(g1) -loaded(g2)

Verify plan? (y/n) y
verifying..
The plan is verified!
yes
| ?- plan(1).

calling sato...
  run time (seconds)                 0.00
no
| ?- plan(2).

calling sato...
  run time (seconds)                 0.00
```

```
calling sato...
  run time (seconds)                 0.00

0.  alive -loaded(g1) -loaded(g2)

ACTIONS: load(g2)

1.  alive  loaded(g2) -loaded(g1)

ACTIONS: shoot(g2)

2. -alive -loaded(g1) -loaded(g2)

Verify plan? (y/n) y
verifying..
The plan is verified!
yes
```

D The Standard $\mathcal{C}$ File: History Mode

The standard $\mathcal{C}$ file consists of three main sections: (i) operator declarations, (ii) macros for translating $\mathcal{C}$ into causal logic, and (iii) declarations of a number of standard sorts and causal laws.

```
% File: 'C.h' -- for use with C in history mode.
% The macro symbol 'maxtime' must be declared before including this file.

%% Operators for the language {\cal C}.
:- op(1010,fx,caused).
:- op(1010,xfx,causes).
:- op(1010,xf,may).
:- op(1010,yfx,cause).
:- op(1020,xfx,if).
:- op(1030,xfx,after).
:- op(1010,fx,inertial).
:- op(1010,fx,exogenous).
:- op(1010,fx,nonexecutable).
:- op(1010,fx,default).
:- op(1010,fx,always).
:- op(1010,fx,never).
:- op(1010,fx,constant).


%% Macros translating {\cal C} propositions into causal laws.  An alternative
%% to the translation given below for static laws would be
%%       caused #1 if #2
%%                -> (h(#2,T) => h(#1,T)).
%% If every fluent has its value at time zero determined exogenously, the two
%% translations are equivalent.
:- macros
```

```
caused #1 if #2 after #3
        -> (next(_N,_T) && h(#3,_N) && h(#2,_T) => h(#1,_T)) ;
caused #1 if #2
        -> ( (h(#2,0) ->> h(#1,0))
           ; (_T>0 && h(#2,_T) => h(#1,_T)) ) ;

caused #1 after #3
        -> caused #1 if true after #3 ;
caused #1
        -> caused #1 if true ;

#1 causes #2 if #3
        -> caused #2 if true after #3 && o(#1) ;
#1 causes #2
        -> caused #2 if true after o(#1) ;

inertial #1,#2
        -> (inertial #1; inertial #2) ;

inertial #1
        -> caused #1 if #1 after #1 ;

exogenous #1,#2
        -> (exogenous #1; exogenous #2) ;

exogenous #1
        -> (caused #1 if #1; caused -(#1) if -(#1)) ;

never #1
        -> caused false if #1 ;
always #1
        -> caused false if -(#1) ;

nonexecutable #1 if #2
        -> caused false after o(#1) && #2 ;
nonexecutable #1
        -> caused false after o(#1) ;

default #1 if #2
        -> caused #1 if #1 && #2 ;
default #1
        -> caused #1 if #1 ;

#1 may cause #2 if #3
        -> caused #2 if #2 after #3 && o(#1) ;
#1 may cause #2
        -> caused #2 if #2 after o(#1) ;

constant #1
        -> ( caused false if #1 after -(#1)
           ; caused false if -(#1) after #1 );
```

```
        h(-(#1),#2)
                -> -h(#1,#2) ;
        h((#1 && #2),#3)
                -> h(#1,#3)  && h(#2,#3) ;
        h((#1  & #2),#3)
                -> h(#1,#3)   & h(#2,#3) ;
        h((#1  , #2),#3)
                -> h(#1,#3)   , h(#2,#3) ;
        h((#1 ++ #2),#3)
                -> h(#1,#3)  ++ h(#2,#3) ;
        h((#1 ->> #2),#3)
                -> h(#1,#3) ->> h(#2,#3) ;
        h((#1 <-> #2),#3)
                -> h(#1,#3) <-> h(#2,#3) ;
        h((/\ #1 : #2),#3)
                -> /\ #1 : h(#2,#3) ;
        h((\/ #1 : #2),#3)
                -> \/ #1 : h(#2,#3) ;

        o(-(#1),#2)
                -> -o(#1,#2) ;
        o((#1 && #2),#3)
                -> o(#1,#3)  && o(#2,#3) ;
        o((#1  & #2),#3)
                -> o(#1,#3)   & o(#2,#3) ;
        o((#1  , #2),#3)
                -> o(#1,#3)   , o(#2,#3) ;
        o((#1 ++ #2),#3)
                -> o(#1,#3)  ++ o(#2,#3) ;
        o((#1 ->> #2),#3)
                -> o(#1,#3) ->> o(#2,#3) ;
        o((#1 <-> #2),#3)
                -> o(#1,#3) <-> o(#2,#3) ;
        o((/\ #1 : #2),#3)
                -> /\ #1 : o(#2,#3) ;
        o((\/ #1 : #2),#3)
                -> \/ #1 : o(#2,#3) ;

        h(o(#1),#2)
                -> o(#1,#2).

%% A few auxilliary macros, some with Prolog 'where' clauses.
:- macros
        #1 : #2 -> h(#2,#1)
                where integer(#1) ;
        #1 - #2 -> #3
                where integer(#1), integer(#2), #3 is #1 - #2 ;
        #1 + #2 -> #3
                where integer(#1), integer(#2), #3 is #1 + #2 .
```

```
%% Standard sort declarations and associated causal laws.
:- sorts
 action;
 fluent
   >> ( ( inertialFalseFluent;
                 inertialTrueFluent    ) >> inertialFluent ;
        ( defaultFalseFluent;
                 defaultTrueFluent     ) >> exogenousFluent
      );
 time >> step.

:- variables
 _N,_N1,_N2 :: step;
 _A,_A1,_A2 :: action;
 _T,_T1,_T2 :: time;
 _F,_F1,_F2 :: fluent;
 _DF :: defaultFalseFluent;
 _DT :: defaultTrueFluent;
 _IF :: inertialFalseFluent;
 _IT :: inertialTrueFluent.

:- constants
 0..maxtime   :: time;
 0..maxtime-1 :: step;
 o(action,step), h(fluent,time) :: atomicFormula.

h(_F,0)  =>  h(_F,0).
-h(_F,0) => -h(_F,0).

next(_N,_T) &&  h(_IT,_N) &&  h(_IT,_T) =>  h(_IT,_T).
next(_N,_T) && -h(_IF,_N) && -h(_IF,_T) => -h(_IF,_T).

_T>0 &&  h(_DT,_T) =>  h(_DT,_T).
_T>0 && -h(_DF,_T) => -h(_DF,_T).

o(_A,_N)  =>  o(_A,_N).
-o(_A,_N) => -o(_A,_N).
```