Missionaries and Cannibals in the Causal Calculator

Vladimir Lifschitz
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712, USA

Abstract

A knowledge representation formalism is "elaboration tolerant" to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances. John McCarthy illustrated this idea by defining 19 elaborations of the Missionaries and Cannibals Problem. We argue that, to a certain degree, the goal of elaboration tolerance is met by the input language of Norman McCain's Causal Calculator. We present formal descriptions of the basic Missionaries and Cannibals Problem and of ten of McCarthy's enhancements as input files accepted by the Causal Calculator. Each enhancement is obtained from the basic formulation by the simplest kind of elaboration—adding postulates.

1 Introduction

This note is about *elaboration tolerance*—a fundamental idea in knowledge representation, described by John McCarthy [9] as follows:

> A formalism is elaboration tolerant to the extent that it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances... Human-level AI will require representations with much more elaboration tolerance than those used by present AI programs, because human-level AI needs to be able to take new phenomena into account.
>
> The simplest kind of elaboration is the addition of new formulas. Next comes changing the values of parameters. Adding new arguments to functions and predicates represents more of a change.

McCarthy illustrates this idea by defining 19 elaborations of the Missionaries and Cannibals Problem (MCP): "Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross?" In one of his elaborations, the boat leaks and must be bailed concurrently with rowing; in others, an oar has to be recovered from the opposite bank of the river, or cannibals can be converted into missionaries, and so forth. The challenge is to design a knowledge representation formalism that would allow us to perform these elaborations by simple means, preferably by just adding new formulas.

This paper argues that, to a certain degree, this goal is met by the input language of the Causal Calculator (CCALC)—a system for query answering and satisfiability planning designed and implemented at the University of Texas.[1] Presented below are formal descriptions of the basic form of MCP and of (certain interpretations of) ten of its enhancements from [9] as input files accepted by CCALC. Each enhancement is obtained from the basic MCP by the simplest kind of elaboration—adding postulates. To stress this fact, the common part of all 11 forms of MCP is "factored out" and placed in a separate file, `common.t`, which is included in each of the input files using the `include` directive of CCALC. The input file for the basic version of MCP, `basic.t`, is very short: essentially the only thing it adds to `common.t` is the statement of the initial conditions and the goal. (The extension `.t` in the names of both files indicates that the files are intended for the use of CCALC in the "transition mode," discussed in the next section.)

[1] `http://www.cs.utexas.edu/users/tag/cc/` .

For each version except one, CCALC has found a solution and verified that the problem could not be solved in fewer steps. Our interest in automatic plan generation is orthogonal to the investigation of elaboration tolerance, which is a property of knowledge representation formalisms—not of algorithms or software systems. But, as a practical matter, it is convenient to have a planner available as a debugging tool.

The main ideas of this paper are not special to the input language of CCALC; other action languages and logic-based planners could be used instead. But CCALC provides a convenient platform for demonstrating these ideas.

After a brief discussion of the Causal Calculator in Section 2, we describe how two ideas related to elaboration tolerance—the use of the abnormality predicate [8] and describing objects in terms of attributes—apply to CCALC (Sections 3–5). In Section 6 a formalization of the basic MCP is presented, and Sections 7–16 describe its elaborations. In Section 17 we comment on some of the remaining challenges.

2 Transition Mode of the Causal Calculator

The original version of CCALC was written by Norman McCain as part of his dissertation [5]; later on, he was assisted by Esra Erdem, Vladimir Lifschitz, Norman Richards, Armando Tacchella and Hudson Turner. Some experimental results on CCALC planning are reported in [7].

In the "transition mode," CCALC operates with a description of a *transition system*—a directed graph whose edges correspond to the transitions caused by the execution of actions. The standard include file `C.t` allows the user to describe transition systems in action language $\mathcal{C}$ from [1].

Here is an example:

```
% File 'example.t'

:- include 'C.t'.

:- sorts
 vessel; loc.

:- variables
 V                  :: vessel;
 L,L1               :: loc.

:- constants
 boat               :: vessel;
 b1, b2             :: loc;
 at(vessel,loc)     :: inertialTrueFluent;
 cross(vessel,loc) :: action.

 cross(V,L) causes at(V,L).
 nonexecutable cross(V,L) if at(V,L).

 always \/ L: at(V,L).
 caused -at(V,L1) if at(V,L) && -(L=L1).

:- plan
facts ::
 0: at(boat,b1);
goals ::
 1: at(boat,b2).
```

This file describes a domain consisting of objects of two kinds: vessels and locations; `V` is a variable for vessels, and `L`, `L1` are variables for locations. There exists one vessel `boat`, and there are two locations—banks `b1`, `b2`.

A state of the transition system defined in this file is characterized by the truth values of the fluents `at(boat,b1)` and `at(boat,b2)`. As discussed below, in each state of the system one of these fluents is true and the other is false, so that only two states are possible: S_1 (`boat` is at `b1`) and S_2 (`boat` is at `b2`).

In this transition system, an action is a subset of the elementary actions `cross(boat,b1)`, `cross(boat,b2)`. The execution of an action is understood as the concurrent execution of the elementary actions that belong to it. However, the first action (crossing to `b1`) is not executable in state S_1, and the second is not executable in state S_2, so that, in this example, the concurrent execution of more than one elementary action is never possible. The empty action—waiting—can be performed in either state.

The sorts `inertialTrueFluent` and `action` are defined in `C.t`. The use of `inertialTrueFluent` rather than `fluent` in the declaration of `at` tells CCALC that this fluent, once true, tends to remain true. The solution to the frame problem incorporated in CCALC is based on the ideas of [10], [11] and [6].

The constant declarations are followed by four propositions. The first two describe the effect of `cross(V,L)` (it causes `at(V,L)` to become true) and its precondition (it is nonexecutable if `at(V,L)` is true). The `always` proposition asserts that, in every state, every vessel is at a certain location (`\/` is the CCALC sign for the existential quantifier, or rather for the corresponding finite disjunction). This proposition tells us that the fluents `at(boat,b1)` and `at(boat,b2)` cannot be simultaneously false. The fourth proposition says that if a vessel `V` is at a certain location then there is a cause for `V` not to be at any other location (in CCALC, `-` denotes negation and `&&` denotes conjunction). This proposition tells us that the fluents `at(boat,b1)` and `at(boat,b2)` cannot be simultaneously true; in addition, it makes `-at(V,L1)` an indirect effect of any action that causes `at(V,L)` with `L` different from `L1`.

The last part of the file describes a planning problem: given that `at(boat,b1)` is true at time 0, make `at(boat,b2)` true at time 1.

Given this input file, CCALC replaces each of the given propositions with its ground instances, compiles the grounded propositions into a set of propositional formulas similar to those used in satisfiability planning [3], and calls a propositional solver (SATO [12] is the default) to find a solution. It produces the output

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

0.  at(boat,b1)

ACTIONS: cross(boat,b2)

1.  at(boat,b2)
```

that shows a one-step plan, along with the fluents that are true at each of the time instants 0, 1 when that plan is executed.

3 Abnormality

In the interests of elaboration tolerance, it is useful to make some propositions of CCALC defeasible. Imagine, for instance, that we want to enhance the boat domain description above by the actions of breaking a vessel into pieces and shipping the pieces to other locations. In such an enhancement, the assertion that every vessel has a location

```
always \/ L: at(V,L)
```

would no longer be valid. An elaboration tolerant formalism should allow us to retract such an assertion, in the spirit of nonmonotonic reasoning, by adding new postulates.

To make a constraint defeasible, we will "label" it, for instance:

```
(l1(V)) always \/ L: at(V,L).
```

One of the macro definitions in the standard file `C.t` expands this line into

```
always -ab(l1(V)) ->> \/ L: at(V,L).
```

(`->>` is material implication). We understand `always` to mean "in all states"; a labeled `always` means "in all states, normally" or "in all states, by default." The symbol `ab` is declared in `C.t` as follows:

```
:- constants
 ab(abLabel) :: defaultFalseFluent.
```

A label has to be declared before it is used:

```
:- constants
 l1(vessel) :: abLabel.
```

To retract the existence of location constraint when the vessel is broken into pieces we would add a proposition like this:

```
caused ab(l1(V)) if broken(V).
```

In the context of MCP, the labeling mechanism is needed, for instance, to formalize McCarthy's Elaboration 17: "if the strongest of the missionaries rows fast enough, the cannibals won't have gotten so hungry that they will eat the missionaries." The assumption that missionaries are never outnumbered by cannibals in file `common.t` is labeled, and then it is partially retracted in file `jmc17.t`.

In our formalizations of MCP, abnormality labels are used with many `always`, `never` and `nonexecutable` propositions. Greater flexibility can be achieved by including many or all free variables of a proposition in its abnormality label as arguments. But here, for simplicity, we use abnormality labels without arguments.

Introducing abnormality labels required some changes in standard file `C.t`, but not in the code of CCALC.

4 Action Attributes

From the perspective of MCP, representing the action of crossing the river by an expression like `cross(boat,b2)` is inadequate: this expression does not provide enough information to decide how crossing affects the numbers of missionaries and cannibals on both banks (unless boarding the boat is treated as a separate action). An expression like `cross(boat,b2,1,1)` (1 missionary and 1 cannibal use the boat to move to Bank 2) would do.

Some of McCarthy's elaborations would require, however, that `cross` be given even more additional arguments. In Elaboration 17 mentioned above, we distinguish between rowing fast and rowing slowly; that would require expressions like `cross(boat,b2,1,1,fast)`. In Elaboration 6, only one missionary and one cannibal can row. In this case, the expression denoting the action should tell us which of the people crossing can row.

As discussed in the introduction, adding arguments to functions and predicates is what we want to avoid: our goal is to perform all elaborations by adding postulates. To achieve this goal, we will think of an action as something that can be described more completely or less completely, depending on how many details about its execution are provided. The action of crossing the river will be denoted by the symbol `cross` without arguments. To say that a vessel `V` is used to cross, we will write `cross_in(V)`. To say that the destination is `L`, we will write `cross_to(L)`. Syntactically, `cross_in(V)` and `cross_to(L)` are action symbols, like `cross`. But we will think of them as merely "attributes," or details of execution, of action `cross`. This intuition will be expressed by the postulates asserting that

- if the action `cross` is executed then there exists a unique `V` such that `cross_in(V)` holds;
- if the action is not executed then `cross_in(V)` does not hold for any `V`;

similarly,

- if the action `cross` is executed then there exists a unique `L` such that `cross_to(L)` holds;
- if the action is not executed then `cross_to(L)` does not hold for any `L`.

New elaborations will involve extending the language by new attribute symbols, instead of adding new arguments to the existing action symbols. The advantage of this method is that the description of the effects of an action given earlier does not have to be modified when an enhancement is introduced. Instead, we add causal laws that describe new effects of the action in terms of its newly introduced attributes.

Technically, using action attributes is similar to "operator splitting" [4, 2]—the representation method that uses expressions like

$$Source(a,3) \wedge Object(b,3) \wedge Destination(c,3)$$

to say that b was moved from a to c at time 3, instead of $Move(a,b,c,3)$. Operator splitting was invented to speed up search. Our use of attributes is motivated by elaboration tolerance.

Here is the example from Section 3 rewritten in terms of attributes:

```
% File 'common1.t': action of crossing
% and two of its attributes

:- include 'C.t'.      % standard file

:- macros
 attribute -> action.

:- sorts
 vessel; loc.

:- variables
 V,V1                 :: vessel;
 L,L1                 :: loc.

:- constants
 at(vessel,loc)       :: inertialTrueFluent;
 cross                :: action;
 cross_in(vessel)     :: attribute;
 cross_to(loc)        :: attribute.

% cross_in and cross_to are attributes
% of action cross
nonexecutable -(cross <-> \/ V: cross_in(V)).
nonexecutable -(cross <-> \/ L: cross_to(L)).
nonexecutable cross_in(V) && cross_in(V1)
                                   if -(V=V1).
nonexecutable cross_to(L) && cross_to(L1)
                                   if -(L=L1).

% Properties of crossing
cross_in(V) && cross_to(L) causes at(V,L).
nonexecutable cross_in(V) && cross_to(L)
                                if at(V,L).


% Properties of at
always \/ L: at(V,L).
caused -at(V,L1) if at(V,L) && -(L=L1).
```

This file does not define any constants and does not contain the planning problem found at the end of `example.t`. The following file includes `common1.t` and contains the two missing components:

```
% 'common1-test.t'

:- include 'common1.t'.

:- constants
 boat                 :: vessel;
 b1, b2               :: loc.

:- plan
facts ::
 0: at(boat,b1);
```

```
goals ::
 1: at(boat,b2).
```

Given `common1-test.t` as input, CCALC produces the following output:

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

0.  at(boat,b1)

ACTIONS: cross cross_in(boat) cross_to(b2)

1.  at(boat,b2)
```

We understand the list

```
cross cross_in(boat) cross_to(b2)
```

in the output of CCALC as follows: The action to be executed is crossing the river; the river is to be crossed in the boat; the destination is the second bank. Future versions of CCALC will possibly format plans expressed in terms of attributes differently, perhaps like this:

```
cross(in:boat, to:b2).
```

They may also allow us express postulates like

```
nonexecutable -(cross <-> \/ V: cross_in(V)).
nonexecutable -(cross <-> \/ L: cross_to(L)).
nonexecutable cross_in(V) && cross_in(V1)
                                 if -(V=V1).
nonexecutable cross_to(L) && cross_to(L1)
                                 if -(L=L1).
```

more concisely—by saying that `in` and `to` are attributes of `cross`.

5 Groups Crossing

To illustrate the use of attributes for achieving the goal of elaboration tolerance, we will enhance `common1.t` by new attributes. For every "group" (such as missionaries or cannibals) we can ask how many members of the group are aboard while the action `cross` is executed. This number is the attribute of `cross` corresponding to the given group. The counting mechanism introduced in this elaboration will be used for many purposes—to count oars in Elaboration 5, to count those who can row in Elaboration 6, and so forth.

```
% 'common2.t': extending 'common1.t' by
% new attributes of crossing

:- include 'common1.t'.

:- sorts
 number; group.

:- variables
 M,N                                 :: number;
 G                                   :: group;
 X                                   :: computed.

:- constants
 0..maxInt                         :: number;
 num(group,loc,number) :: inertialTrueFluent;
 cross_howmany(group,number) :: attribute.

% cross_howmany is a family of attributes
nonexecutable
    -(cross <-> \/ N: cross_howmany(G,N)).
nonexecutable
    cross_howmany(G,M) && cross_howmany(G,N)
    if -(M=N).

% addition and subtraction in the
% range 0..maxInt
:- macros
 sum(#1,#2,#3) ->
         #1 is min((#2)+(#3),maxInt);
 diff(#1,#2,#3) ->
         #1 is max((#2)-(#3),0).

% properties of crossing described in
% terms of the new attributes
cross_in(V) && cross_howmany(G,M)
   causes num(G,L,X)
   if at(V,L) && num(G,L,N) && diff(X,N,M).
cross_in(V) && cross_to(L)
            && cross_howmany(G,M)
              causes num(G,L,X)
              if num(G,L,N) && sum(X,N,M).
nonexecutable
   cross_in(V) && cross_howmany(G,M)
   if at(V,L) && num(G,L,N) && M>N.

% properties of num
always \/ N: num(G,L,N).
caused -num(G,L,N) if num(G,L,M) && -(M=N).
```

(By declaring `X` to be a `computed` variable we tell CCALC that, in the process of grounding, there will be no need to select values for `X`; they will have been always pre-computed.)

The propositions describing the additional effects of action `cross` say that if the action is performed by `M` members of a group then the number of members of the group at the departure location decreases by `M`, and the number of members of the group at destination increases by M. This assertion is correct only if `cross` is not executed concurrently with any other action that affects these two numbers. This is an essential limitation of the representation used in `common2.t`. The basic MCP and most of its elaborations introduced in [9] do not allow such "difficult" concurrency (most of them involve no concurrency at all), but there are exceptions. We will return to this question several times.

To test the new theory of crossing, we'll assume that there are 3 people initially on one bank and 2 on the other, and ask CCALC to move everyone to the second bank in one step:

```
% 'common2-test.t'

:- macros
 maxInt -> 6.

:- include 'common2.t'.

:- constants
 boat               :: vessel;
 b1, b2             :: loc;
 person             :: group.

:- plan
facts ::
 0: at(boat,b1),
 0: num(person,b1,3),
 0: num(person,b2,2);
goals ::
 1: num(person,b1,0).
```

The output:

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

0.  at(boat,b1)  num(person,b1,3)
num(person,b2,2)

ACTIONS: cross cross_in(boat)
cross_to(b2) cross_howmany(person,3)

1.  at(boat,b2)  num(person,b1,0)
num(person,b2,5)
```

6 Basic MCP

Now we are ready to show file `common.t` mentioned in the introduction—the file included in all 11 versions of MCP.

```
% 'common.t': common for all forms of MCP

:- include 'common2.t'.

:- variables
 K                       :: number.

:- constants
 boat                    :: vessel;
 b1, b2                  :: loc;
 mi,ca                   :: group;
 capacity(vessel,number) :: exogenousFluent.

:- macros
% #1 missionaries are outnumbered
% by #2 cannibals
 outnumbered(#1,#2)
      -> (#2 > #1) && (#1 > 0).

% labels to be used for reference if we
% decide to retract propositions below
:- constants
  1..5 :: abLabel.

% missionaries should not be outnumbered
% in any location
(1) never num(mi,L,M) && num(ca,L,N)
                      && outnumbered(M,N).

% additional preconditions for crossing

% someone should be in the boat
(2) nonexecutable cross_howmany(mi,0)
                  && cross_howmany(ca,0).

% but not too many
(3) nonexecutable
     cross_in(V) && cross_howmany(mi,M)
                 && cross_howmany(ca,N)
   if capacity(V,K) && sum(X,M,N) && X>K.

% missionaries should not be outnumbered
% on the way
(4) nonexecutable cross_howmany(mi,M)
                  && cross_howmany(ca,N)
    if outnumbered(M,N).

% boat capacity
always
    capacity(V,M) && capacity(V,N) ->> M=N.
(5) always capacity(boat,2).
```

The basic form of MCP can be formalized as follows:

```
% 'basic.t': original MCP
```

```
:- macros
 maxInt -> 3.

:- include 'common.t'.

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 10..11: (num(mi,b1,0) && num(ca,b1,0)).
```

The expression `10..11` in the last line instructs CCALC to try to find a plan of length 10 and then, if there is no such plan, try length 11. Since the length of the shortest plan for the basic form of MCP is actually 11, CCALC finds such a plan after determining that a shorter plan is impossible. The output begins with

```
calling sato...
  No plan of length 10,
run time (seconds)                 6.84
calling sato...
run time (seconds)                17.59
```

which is followed by the usual 11 step solution.

7 Boat Can Carry Three

The boat can carry three. Then four pairs can cross.

The assumption in `common.t` that the boat can only carry two has an abnormality label. This fact allows us to retract it:

```
% 'jmc4.t': McCarthy's Elaboration 4

:- macros
 maxInt -> 4.

:- include 'common.t'.

% retract (5) from 'common.t'

caused ab(5).

:- constants
 6 :: abLabel.

(6) always capacity(boat,3).

:- plan
facts ::
 0: (num(mi,b1,4) && num(ca,b1,4)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 8..9: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 7 and 18 seconds.

8 An Oar on each Bank

"There is an oar on each bank. One person can cross in the boat with just one oar, but two oars are needed if the boat is to carry two people. We can send a cannibal to get the oar and then we are reduced to the original problem" [9].

We'll count oars using the group mechanism used above to count missionaries and cannibals.

```
% 'jmc5.t': McCarthy's Elaboration 5

:- macros
 maxInt -> 3.

:- include 'common.t'.

:- constants
 oars        :: group;
 6,7         :: abLabel.

(6) nonexecutable
  cross_in(boat) && cross_howmany(oars,0).
(7) nonexecutable
 cross_in(boat) && cross_howmany(oars,1) &&
 cross_howmany(mi,M) && cross_howmany(ca,N)
    if M+N > 1.

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)
     && num(oars,b1,1)),
 0: (num(mi,b2,0) && num(ca,b2,0)
     && num(oars,b2,1)),
 0: at(boat,b1);
goals ::
 12..13: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 27 and 44 seconds.

9 Not Everyone Can Row

Only one missionary and one cannibal can row. The problem is still solvable, but now the length of the shortest plan is 13.

In addition to the group `mi` of missionaries, we introduce its subset rowers(mi), and similarly for cannibals. To avoid the possibility of iterating function rowers (which would create an infinite set of ground terms), we limit it to "basic groups."

The fact that any basic group BG is a superset of rowers(BG) is reflected in the postulates that prohibit the states in which the number of members of BG in any location is smaller than the number of members of rowers(BG) in the same location, as well as the actions in which the number of members of BG crossing the river is smaller than the number of members of rowers(BG) crossing.

```
% 'jmc6.t':  McCarthy's Elaboration 6

:- macros
 maxInt -> 3.

:- include 'common.t'.

:- sorts
 group >> basicGroup.

:- variables
 BG                    :: basicGroup.

:- constants
 mi,ca                 :: basicGroup;
 rowers(basicGroup)    :: group.

never
 num(rowers(BG),L,M) && num(BG,L,N) && M>N.

nonexecutable cross_howmany(rowers(BG),M)
             && cross_howmany(BG,N) if M>N.

:- constants
 6 :: abLabel.

(6) nonexecutable
             cross_howmany(rowers(mi),0)
             && cross_howmany(rowers(ca),0).

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)),
 0: (num(rowers(mi),b1,1)
             && num(rowers(ca),b1,1)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 12..13: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 286 and 273 seconds.

10 A Very Big Cannibal

The biggest cannibal cannot fit in the boat with another person. The problem is still solvable, but now the length of the shortest plan turns out to be 15.

We can keep track of the position of the biggest cannibal using the group mechanism introduced in Section 5 if we treat him as a one-person group.

```
% 'jmc8.t':  McCarthy's Elaboration 8

:- macros
 maxInt -> 3.

:- include 'common.t'.

:- constants
 vbc :: group. % very big cannibal

never num(vbc,L,M) && num(ca,L,N) && M>N.

nonexecutable cross_howmany(vbc,M)
              && cross_howmany(ca,N) && M>N.

:- constants
 6 :: abLabel.

(6) nonexecutable cross_howmany(vbc,1)
           && cross_howmany(mi,M)
           && cross_howmany(ca,N) if M+N>1.

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)
                     && num(vbc,b1,1)),
 0: (num(mi,b2,0) && num(ca,b2,0)
                     && num(vbc,b2,0)),
 0: at(boat,b1);
goals ::
 14..15: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 2149 and 9746 seconds.

11 Big Cannibal and Small Missionary

If the biggest cannibal is isolated with the smallest missionary—in the boat or on either bank—the latter will be eaten. A plan that works for the basic MCP can be adapted to this enhancement if we specify, for every crossing, whether the biggest cannibal and the smallest missionary are aboard.

```
% 'jmc9.t':  McCarthy's Elaboration 9
:- macros
```

```
 maxInt -> 3.

:- include 'common.t'.

:- constants
 bca :: group;        % big cannibal
 smi :: group.        % small missionary

never num(bca,L,M) && num(ca,L,N) && M>N.

never num(smi,L,M) && num(mi,L,N) && M>N.

nonexecutable cross_howmany(bca,M)
&& cross_howmany(ca,N) && M>N.

nonexecutable cross_howmany(smi,M)
        && cross_howmany(mi,N) && M>N.

:- constants
 6,7 :: abLabel.

(6) never num(bca,L,1) && num(smi,L,1)
                       && num(mi,L,1).
(7) nonexecutable cross_howmany(bca,1)
              && cross_howmany(smi,1).

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)
 && num(bca,b1,1) && num(smi,b1,1)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 10..11: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 18 and 22 seconds.

12 Converting Cannibals

Three missionaries alone with a cannibal can convert him into a missionary. We'll treat converting as an action that can be included in the plan, rather than an event that happens whenever the preconditions are satisfied.

Do we allow crossing the river and converting a cannibal to occur in parallel? Can a solution begin, for instance, with two cannibals crossing to Bank 2 while the third cannibal is being converted into a missionary on Bank 1? If yes, this is an example of "difficult" concurrency (Section 5) that the approach of this paper does not allow.

Let's assume that the concurrent execution of the two actions involved is not allowed in a plan. Even so, the problem can be solved in 10 steps, including one conversion, instead of 11 steps required for the basic MCP.

```
% 'jmc11.t': McCarthy's Elaboration 11

:- macros
 maxInt -> 6.

:- include 'common.t'.

:- constants
 convert          :: action;
 convert_at(loc)  :: attribute.

% convert_at is an attribute of convert
nonexecutable -(convert
             <-> \/ L: convert_at(L)).
nonexecutable convert_at(L) && convert_at(L1)
             if -(L=L1).
convert_at(L) causes num(mi,L,X)
             if num(mi,L,N) && sum(X,N,1).
convert_at(L) causes num(ca,L,X)
             if num(ca,L,N) && diff(X,N,1).

:- constants
 6,7 :: abLabel.

(6) nonexecutable convert_at(L)
             if num(mi,L,N) && N<3.
(7) nonexecutable convert_at(L)
             if num(ca,L,N) && -(N=1).

% no concurrent actions
nonexecutable cross && convert.

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 9..10: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 37 and 55 seconds.

13 The Bridge

"There is a bridge. This makes it obvious to a person that any number can cross provided two people can cross at once... There is no need to get rid of the boat unless this is a part of the elaboration wanted" [9]. In the formalization below we do not get rid of the boat, but using the bridge and the boat concurrently is prohibited to avoid "difficult" concurrency.

```
% 'jmc13.t':  McCarthy's Elaboration 13

:- macros
 maxInt -> 5.

:- include 'common.t'.

:- constants
 useBridge                :: action;
 useBridge_from(loc)      :: attribute;
 useBridge_to(loc)        :: attribute;
 useBridge_howmany(group,number)
                          :: attribute.

% useBridge_from, useBridge_to and
% useBridge_howmany are attributes:

 nonexecutable -(useBridge
        <-> \/ L: useBridge_from(L)).
 nonexecutable -(useBridge
        <-> \/ L: useBridge_to(L)).
 nonexecutable -(useBridge
        <-> \/ N: useBridge_howmany(G,N)).
 nonexecutable useBridge_from(L)
        && useBridge_from(L1) if -(L=L1).
 nonexecutable useBridge_to(L)
        && useBridge_to(L1) if -(L=L1).
 nonexecutable useBridge_howmany(G,M)
        && useBridge_howmany(G,N) if -(M=N).

% properties of using bridge
useBridge_from(L) && useBridge_howmany(G,M)
        causes num(G,L,X)
        if num(G,L,N) && diff(X,N,M).
useBridge_to(L) && useBridge_howmany(G,M)
        causes num(G,L,X)
        if num(G,L,N) && sum(X,N,M).
nonexecutable useBridge_from(L)
        && useBridge_howmany(G,M)
        if num(G,L,N) && M>N.
nonexecutable useBridge_from(L)
        && useBridge_to(L1)
        if -((L=b1 && L1=b2)
             ++ (L=b2 && L1=b1)).

:- constants
 6 :: abLabel.

(6) nonexecutable useBridge_howmany(mi,M)
     && useBridge_howmany(ca,N) if M+N > 2.

% no concurrent actions
nonexecutable cross && useBridge.

:- plan
facts ::
 0: (num(mi,b1,5) && num(ca,b1,5)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 4..5: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 2 seconds each time.

14 The Boat Leaks

The boat leaks and must be bailed concurrently with rowing. To make the problem more interesting, we assume that bailing is only needed when two persons are crossing, and that one person would not be able to row and to bail simultaneously.

In this example concurrency is essential, but it does not present the problem discussed in Section 5: bailing does not change the numbers of missionaries and cannibals in any location.

```
% 'jmc14.t': McCarthy's Elaboration 14

:- macros
 maxInt -> 3.

:- include 'common.t'.

:- constants
 bail        :: action;
 6,7         :: abLabel.

(6) nonexecutable cross_howmany(mi,M)
    && cross_howmany(ca,N) && -bail
                   if sum(X,M,N) && X>1.
(7) nonexecutable cross_howmany(mi,M)
    && cross_howmany(ca,N) && bail
                   if sum(X,M,N) && X=1.

:- plan
facts ::
 0: (num(mi,b1,3) && num(ca,b1,3)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 10..11: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 7 and 9 seconds.

15 The Island

There is an island. Then any number can cross.

Formalizing this enhancement is straightforward: we just need to declare `island` to be an additional location. But computationally the problem becomes too difficult for CCALC (or rather for SATO), even if we assume only 4 missionaries and 4 cannibals. To make the computational task easier, we modify the goal as follows: one missionary should stay on Bank 1, one cannibal should be on the island, and the rest of the company, along with the boat, should reach Bank 2. Once this subgoal is achieved, the original goal can be achieved by sending a cannibal to bring the missing missionary and then the missing cannibal to Bank 2 (4 moves).

The output of CCALC shows that the modified planning problem can be solved in 11 steps. Consequently, the original problem can be solved in 15 steps.

```
% 'jmc16.t':  McCarthy's Elaboration 16

:- macros
 maxInt -> 4.

:- include 'common.t'.

:- constants
 island :: loc.

:- plan
facts ::
 0: (num(mi,b1,4) && num(ca,b1,4)),
 0: (num(mi,island,0) && num(ca,island,0)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
 10..11: (num(mi,b1,1) && num(ca,b1,0)
     && num(mi,island,0) && num(ca,island,1)
     && at(boat,b2)).
```

SATO run times: 192 and 1894 seconds.

16 Cannibals Are Not Hungry

"There are four cannibals and four missionaries, but if the strongest of the missionaries rows fast enough, the cannibals won't have gotten so hungry that they will eat the missionaries. This could be made precise in various ways, but the information is usable even in vague form" [9].

We assume that rowing fast is done so fast that, even when executed many times, it doesn't give cannibals enough time to get hungry; rowing slowly is so slow that cannibals get hungry after you do it just once.

```
% 'jmc17.t':  McCarthy's Elaboration 17

:- macros
 maxInt -> 4.

:- include 'common.t'.

:- sorts
 speed.

:- variables
 S, S1             :: speed.

:- constants
 slowly, fast  :: speed;
 strmi         :: group;
 hungry        :: defaultTrueFluent;
 cross_howfast(speed) :: attribute.

% cross_howfast is an attribute
nonexecutable -(cross <->
      \/ S: cross_howfast(S)).
nonexecutable cross_howfast(S)
      && cross_howfast(S1) if -(S=S1).
never num(strmi,L,M) && num(mi,L,N) && M>N.
nonexecutable cross_howmany(strmi,M) &&
       cross_howmany(mi,N) && M>N.

:- constants
 6 :: abLabel.

(6) nonexecutable cross_howfast(fast)
                 && cross_howmany(strmi,0).

% retract (1) from 'common.t'
caused ab(1) if -hungry.

% retract (4) from 'common.t'
caused ab(4) if -hungry.

cross_howfast(fast) causes -hungry
                    if -hungry.

:- plan
facts ::
 0: -hungry,
 0: (num(mi,b1,4) && num(strmi,b1,1)
     && num(ca,b1,4)),
 0: (num(mi,b2,0) && num(ca,b2,0)),
 0: at(boat,b1);
goals ::
```

```
12..13: (num(mi,b1,0) && num(ca,b1,0)).
```

SATO run times: 1006 and 7361 seconds.

17 Conclusion

In this paper we argue that the input language of CCALC provides a partial solution to the problem of elaboration tolerance in representing actions. Our examples use two innovations in the methodology of representing knowledge in CCALC—abnormality labels and describing actions in terms of their attributes.

We have presented here formalizations of 10 out of the 19 elaborations of MCP described in [9]. Some of the remaining nine present interesting topics for future work.

One significant limitation of our approach to MCP is mentioned in Section 5: our description of the additional effects of crossing on the numbers of missionaries and cannibals at various locations presupposes that actions affecting these numbers are not executed in parallel. This limitation prevented us from allowing concurrency in the discusion of Elaboration 11 (converting cannibals) and Elaboration 13 (the bridge). It has also prevented us from formalizing McCarthy's Elaboration 10: One of the missionaries is Jesus Christ; four can cross. Walking on water will help only if performed concurrently with crossing by boat, and this is an example of "difficult" concurrency. It should be stressed that this difficulty is not in any way inherent in the language of CCALC; see the section on interaction between concurrent actions in [1].

In Elaboration 19 there are two sets of missionaries and cannibals, each with one boat, too far apart along the river to interact. To allow the two boats to be used concurrently, we would need to modify the representation of the `cross` action in terms of attributes used in this note: `cross` would have to be given an argument.

Challenges of different kinds are found in Elaboration 12 (the use of probabilities) and Elaboration 15 (splitting an event into parts).

Acknowledgements

A representation of MCP related to the work described above has been investigated by Mary Heidt. Esra Erdem, Michael Gelfond, Joohyung Lee, John McCarthy, Emilio Remolina and Hudson Turner provided useful comments on earlier drafts of this note. Norman McCain helped with the design of abnormality labels. Thanks to all of them for help and for sharing their ideas with me.

This work was partially supported by National Science Foundation under grant IIS-9732744.

References

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

[2] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Principles of Knowledge Representation and Reasoning: Proc. of the Fifth Int'l Conf.*, pages 374–384, 1996.

[3] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363, 1992.

[4] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *Proc. AAAI-96*, pages 1194–1201, 1996.

[5] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[2] PhD thesis, University of Texas at Austin, 1997.

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

[7] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proc. Sixth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 212–223, 1998.

[8] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[9] John McCarthy. Elaboration tolerance.[3] In progress, 1999.

[10] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[11] Hudson Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[12] Hantao Zhang. An efficient propositional prover. In *Proc. CADE-97*, 1997.

[2] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.Z` .

[3] `http://www-formal.stanford.edu/jmc/elaboration.html` .