

Appendix

The appendix is organized as follows. Section A shows an example run for the basic Missionaries and Cannibals problem. Section B includes all prompts used in LLM+AL. Section C shows the prompts used for the baseline LLM along with CHATGPT-4+CODE. Section D shows example error cases. Section E lists all the Missionaries and Cannibals elaborations that we use. All outputs by the baseline LLMs can be found in the code repository.

All experiments with LLM+AL access model o1-preview through OpenAI's API ("o1-preview"). All experiments are done on a machine with a I9-9900k processor, Nvidia Geforce RTX 2080 Ti, and 32 GB of RAM. Running LLM+AL is mostly bottlenecked by calling the LLM, though the BC+ reasoner occasionally requires significant time.

A The Pipeline Detailed with Missionaries and Cannibals Example

A.1 Input

Example Problem Input

Three missionaries and three cannibals shall cross a river in a boat which carries at most two people at once. The boat cannot cross the river with nothing on it. On either side, the cannibals may never outnumber the missionaries or the missionaries will be eaten. Also, the cannibals may never outnumber the missionaries on the boat or the missionaries will be eaten.

There are four types of things: "vessel", "location", "group", and "integer".
boat is a vessel.

bank1 and bank2 are locations.
missionaries and cannibals are groups.
0..10 are integers.

Actions

Crossing a vessel is an action.

Query

Three missionaries and three cannibals are at location bank1. The boat is at location bank1. Find a plan so that all missionaries and all cannibals are at location bank2.

A.2 BC+ Signature Generation

Actions:

- cross(Vessel) – Cross the river in the Vessel.

Constants:

- loc(Vessel) – The location of the Vessel, remains the same unless explicitly changed, so it is an inertial fluent.
- numOnBank(Location, Group) – The number of people of Group at a Location (bank), remains the same unless explicitly changed, so it is an inertial fluent.
- numCrossing(Vessel, Group) – The number of people of Group crossing in the Vessel during the crossing action, an attribute of `cross(Vessel)`.

BC+ Signature:

```
:= sorts
    vessel;
    location;
    group;
    integer.

:= objects
    boat :: vessel;
    bank1, bank2 :: location;
    missionaries, cannibals :: group;
    0..10 :: integer.

:= variables
    V, V1, V2, V3, V4 :: vessel;
    L, L1, L2, L3, L4 :: location;
    G, G1, G2, G3, G4 :: group;
    N, N1, N2, N3, N4 :: integer;
    Bool, Bool1, Bool2, Bool3, Bool4 :: boolean.
```

```

:- constants
cross(vessel) :: exogenousAction;
numCrossing(vessel, group) :: attribute(integer) of cross(vessel);
loc(vessel) :: inertialFluent(location);
numOnBank(location, group) :: inertialFluent(integer).

```

A.3 Knowledge Generation

```

% Crossing the river in the vessel causes the location of the vessel to change to the opposite bank.
% Crossing the river in the vessel causes the number of people of each group on the departure bank to decrease
    by the number of people of that group crossing in the vessel.
% Crossing the river in the vessel causes the number of people of each group on the arrival bank to increase by
    the number of people of that group crossing in the vessel.
% The vessel cannot cross the river with zero people on it.
% The vessel cannot carry more than two people at once.
% The number of people of each group crossing in the vessel cannot exceed the number of people of that group on
    the bank where the vessel is located.
% The number of people of each group crossing in the vessel cannot be negative.
% After crossing, on either bank, the number of cannibals cannot exceed the number of missionaries unless the
    number of missionaries on that bank is zero.
% During crossing, the number of cannibals crossing in the vessel cannot exceed the number of missionaries
    crossing in the vessel unless the number of missionaries crossing is zero.
% The number of people of each group on either bank cannot be negative or exceed three.

```

A.4 Rule and Query Generation

At this point, $\mathcal{BC}+$ rules are generated based on the previously generated and revised knowledge:

```

% constraints/rules

% 1. Crossing the river in the vessel causes the location of the vessel to change to the opposite bank.
cross(V) causes loc(V) = bank2 if loc(V) = bank1.
cross(V) causes loc(V) = bank1 if loc(V) = bank2.

% 2. Crossing the river in the vessel causes the number of people of each group on the departure bank to
    decrease by the number of people of that group crossing in the vessel.
cross(V) decrements numOnBank(L, G) by N if numCrossing(V, G) = N & loc(V) = L.

% 3. Crossing the river in the vessel causes the number of people of each group on the arrival bank to increase
    by the number of people of that group crossing in the vessel.
% 3.a If the vessel is at bank1, crossing increments numOnBank(bank2, G) by N.
cross(V) increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V) = bank1.
% 3.b If the vessel is at bank2, crossing increments numOnBank(bank1, G) by N.
cross(V) increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V) = bank2.

% 4. The vessel cannot cross the river with zero people on it.
nonexecutable cross(V) if N1 + N2 = 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 5. The vessel cannot carry more than two people at once.
nonexecutable cross(V) if N1 + N2 > 2 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 6. The number of people of each group crossing in the vessel cannot exceed the number of people of that group
    on the bank where the vessel is located.
nonexecutable cross(V) if N1 > N2 & N1 = numCrossing(V, G) & N2 = numOnBank(L, G) & loc(V) = L.

% 7. The number of people of each group crossing in the vessel cannot be negative.
nonexecutable cross(V) if N < 0 & N = numCrossing(V, G).

% 8. After crossing, on either bank, the number of cannibals cannot exceed the number of missionaries unless the
    number of missionaries on that bank is zero.
impossible numOnBank(L, cannibals) > numOnBank(L, missionaries) & numOnBank(L, missionaries) > 0.

% 9. During crossing, the number of cannibals crossing in the vessel cannot exceed the number of missionaries
    crossing in the vessel unless the number of missionaries crossing is zero.

```

```

nonexecutable cross(V) if N2 > N1 & N1 > 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals) .

% 10. The number of people of each group on either bank cannot be negative or exceed three.
impossible numOnBank(L, G) < 0.
impossible numOnBank(L, G) > 3.

% query
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & numOnBank(bank2, missionaries) = 0
        & numOnBank(bank2, cannibals) = 0 & loc(boat) = bank1;
    maxstep: numOnBank(bank2, missionaries) = 3 & numOnBank(bank2, cannibals) = 3.

```

A.5 Initial $\mathcal{BC}+$ Program

After the rule and query generation phase, an initial $\mathcal{BC}+$ program is generated as follows:

```

:- sorts
    vessel;
    location;
    group;
    integer.

:- objects
    boat :: vessel;
    bank1, bank2 :: location;
    missionaries, cannibals :: group;
    0..10 :: integer.

:- variables
    V, V1, V2, V3, V4 :: vessel;
    L, L1, L2, L3, L4 :: location;
    G, G1, G2, G3, G4 :: group;
    N, N1, N2, N3, N4 :: integer;
    Bool, Bool1, Bool2, Bool3, Bool4 :: boolean.

:- constants
    cross(vessel) :: exogenousAction;
    numCrossing(vessel, group) :: attribute(integer) of cross(vessel);
    loc(vessel) :: inertialFluent(location);
    numOnBank(location, group) :: inertialFluent(integer).

% constraints/rules

% 1. Crossing the river in the vessel causes the location of the vessel to change to the opposite bank.
cross(V) causes loc(V) = bank2 if loc(V) = bank1.
cross(V) causes loc(V) = bank1 if loc(V) = bank2.

% 2. Crossing the river in the vessel causes the number of people of each group on the departure bank to
%     decrease by the number of people of that group crossing in the vessel.
cross(V) decrements numOnBank(L, G) by N if numCrossing(V, G) = N & loc(V) = L.

% 3. Crossing the river in the vessel causes the number of people of each group on the arrival bank to increase
%     by the number of people of that group crossing in the vessel.
% 3.a If the vessel is at bank1, crossing increments numOnBank(bank2, G) by N.
cross(V) increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V) = bank1.
% 3.b If the vessel is at bank2, crossing increments numOnBank(bank1, G) by N.
cross(V) increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V) = bank2.

% 4. The vessel cannot cross the river with zero people on it.
nonexecutable cross(V) if N1 + N2 = 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

```

```

% 5. The vessel cannot carry more than two people at once.
nonexecutable cross(V) if N1 + N2 > 2 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals) .

% 6. The number of people of each group crossing in the vessel cannot exceed the number of people of that group
on the bank where the vessel is located.
nonexecutable cross(V) if N1 > N2 & N1 = numCrossing(V, G) & N2 = numOnBank(L, G) & loc(V) = L.

% 7. The number of people of each group crossing in the vessel cannot be negative.
nonexecutable cross(V) if N < 0 & N = numCrossing(V, G) .

% 8. After crossing, on either bank, the number of cannibals cannot exceed the number of missionaries unless the
number of missionaries on that bank is zero.
impossible numOnBank(L, cannibals) > numOnBank(L, missionaries) & numOnBank(L, missionaries) > 0.

% 9. During crossing, the number of cannibals crossing in the vessel cannot exceed the number of missionaries
crossing in the vessel unless the number of missionaries crossing is zero.
nonexecutable cross(V) if N2 > N1 & N1 > 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals) .

% 10. The number of people of each group on either bank cannot be negative or exceed three.
impossible numOnBank(L, G) < 0.
impossible numOnBank(L, G) > 3.

% query
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & numOnBank(bank2, missionaries) = 0
        & numOnBank(bank2, cannibals) = 0 & loc(boat) = bank1;
    maxstep: numOnBank(bank2, missionaries) = 3 & numOnBank(bank2, cannibals) = 3.

```

A.6 Self-Revision: Final \mathcal{BC} + Program

Satisfiability Check After running the initial program, the \mathcal{BC} + reasoner does not compile the program due to a syntax error. The following feedback is provided:

```
% Query 0: Check satisfiability of program signature and rules (ignores other queries). (satisfiable)
:- query = sat.

Cplus2ASP Output:

ERROR: temp.bc:39:21]: The head of an additive law (G in "F decrements G") must be an additive constant (
additiveFluent or additiveAction) .

Since this fails, check the error message(s) .
```

The LLM corrects this issue, re-writing the program as follows. The constant numOnBank is now correctly declared as an additive fluent.

```
% BC+ signature
:- sorts
    vessel;
    location;
    group;
    integer.

:- objects
    boat :: vessel;
    bank1, bank2 :: location;
    missionaries, cannibals :: group;
    0..10 :: integer.

:- variables
    V, V1, V2, V3, V4 :: vessel;
    L, L1, L2, L3, L4 :: location;
    G, G1, G2, G3, G4 :: group;
    N, N1, N2, N3, N4 :: integer;
```

```

Bool, Bool1, Bool2, Bool3, Bool4 :: boolean.

:- constants
cross(vessel) :: exogenousAction;
numCrossing(vessel, group) :: attribute(integer) of cross(vessel);
loc(vessel) :: inertialFluent(location);
numOnBank(location, group) :: additiveFluent(integer);

% Constraints/rules

% 1. Crossing the river in the vessel causes the location of the vessel to change to the opposite bank.
cross(V) causes loc(V) = bank2 if loc(V) = bank1.
cross(V) causes loc(V) = bank1 if loc(V) = bank2.

% 2. Crossing the river in the vessel causes the number of people of each group on the departure bank to
% decrease by the number of people of that group crossing in the vessel.
cross(V) decrements numOnBank(L, G) by N if numCrossing(V, G) = N & loc(V) = L.

% 3. Crossing the river in the vessel causes the number of people of each group on the arrival bank to increase
% by the number of people of that group crossing in the vessel.
% 3.a If the vessel is at bank1, crossing increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V)
% = bank1.
cross(V) increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V) = bank1.

% 3.b If the vessel is at bank2, crossing increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V)
% = bank2.
cross(V) increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V) = bank2.

% 4. The vessel cannot cross the river with zero people on it.
nonexecutable cross(V) if N1 + N2 = 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 5. The vessel cannot carry more than two people at once.
nonexecutable cross(V) if N1 + N2 > 2 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 6. The number of people of each group crossing in the vessel cannot exceed the number of people of that group
% on the bank where the vessel is located.
nonexecutable cross(V) if N1 > N2 & N1 = numCrossing(V, G) & N2 = numOnBank(L, G) & loc(V) = L.

% 7. The number of people of each group crossing in the vessel cannot be negative.
nonexecutable cross(V) if N < 0 & N = numCrossing(V, G).

% 8. After crossing, on either bank, the number of cannibals cannot exceed the number of missionaries unless the
% number of missionaries on that bank is zero.
impossible N1 > N2 & N2 > 0 & N1 = numOnBank(L, cannibals) & N2 = numOnBank(L, missionaries).

% 9. During crossing, the number of cannibals crossing in the vessel cannot exceed the number of missionaries
% crossing in the vessel unless the number of missionaries crossing is zero.
nonexecutable cross(V) if N2 > N1 & N1 > 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 10. The number of people of each group on either bank cannot be negative or exceed three.
impossible N < 0 & N = numOnBank(L, G).
impossible N > 3 & N = numOnBank(L, G).

% Main query
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & numOnBank(bank2, missionaries) = 0
        & numOnBank(bank2, cannibals) = 0 & loc(boat) = bank1;
    maxstep: numOnBank(bank2, missionaries) = 3 & numOnBank(bank2, cannibals) = 3.

```

Sample Query Generation After the satisfiability check, the LLM generates the following sample queries.

```

% Query 1: Have one missionary and one cannibal cross from bank1 to bank2 in the boat (satisfiable)
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;

```

```

0: cross(boat) & numCrossing(boat, missionaries) = 1 & numCrossing(boat, cannibals) = 1.

% Query 2: Try to have two missionaries and one cannibal cross from bank1 to bank2 in the boat (unsatisfiable)
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
    0: cross(boat) & numCrossing(boat, missionaries) = 2 & numCrossing(boat, cannibals) = 1.

% Query 3: Have one cannibal cross from bank1 to bank2 alone in the boat (satisfiable)
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
    0: cross(boat) & numCrossing(boat, missionaries) = 0 & numCrossing(boat, cannibals) = 1.

% Query 4: Try to have one missionary cross from bank1 to bank2 alone in the boat (unsatisfiable)
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
    0: cross(boat) & numCrossing(boat, missionaries) = 1 & numCrossing(boat, cannibals) = 0.

% Query 5: Try to have the boat cross from bank1 to bank2 with no people in it (unsatisfiable)
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
    0: cross(boat) & numCrossing(boat, missionaries) = 0 & numCrossing(boat, cannibals) = 0.

```

Sample and Main Query Feedback And the feedback from running these queries, along with the main query is shown to the LLM as follows.

```

% Query 1: Have one missionary and one cannibal cross from bank1 to bank2 in the boat (satisfiable)
:- query
    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
    0: cross(boat) & numCrossing(boat, missionaries) = 1 & numCrossing(boat, cannibals) = 1.

```

Cplus2ASP Output:

Solving...
Solution: 1

0: loc(boat)=bank1 numOnBank(bank1, cannibals)=3 numOnBank(bank1, missionaries)=3 numOnBank(bank2,
cannibals)=0 numOnBank(bank2, missionaries)=0

ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=1

1: loc(boat)=bank2 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=2 numOnBank(bank2,
cannibals)=1 numOnBank(bank2, missionaries)=1

SATISFIABLE
Models : 1

```

% Query 2: Try to have two missionaries and one cannibal cross from bank1 to bank2 in the boat (unsatisfiable)
:- query

```

```

    0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
        bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
    0: cross(boat) & numCrossing(boat, missionaries) = 2 & numCrossing(boat, cannibals) = 1.

```

Cplus2ASP Output:

Solving...
UNSATISFIABLE
Models : 0
No solution.

```
% Query 3: Have one cannibal cross from bank1 to bank2 alone in the boat (satisfiable)
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
    bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 0 & numCrossing(boat, cannibals) = 1.
```

Cplus2ASP Output:

Solving...

Solution: 1

```
0: loc(boat)=bank1 numOnBank(bank1, cannibals)=3 numOnBank(bank1, missionaries)=3 numOnBank(bank2,
cannibals)=0 numOnBank(bank2, missionaries)=0
```

```
ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0
```

```
1: loc(boat)=bank2 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=3 numOnBank(bank2,
cannibals)=1 numOnBank(bank2, missionaries)=0
```

SATISFIABLE

Models : 1

```
% Query 4: Try to have one missionary cross from bank1 to bank2 alone in the boat (unsatisfiable)
```

```
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
    bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 1 & numCrossing(boat, cannibals) = 0.
```

Cplus2ASP Output:

Solving...

UNSATISFIABLE

Models : 0

No solution.

```
% Query 5: Try to have the boat cross from bank1 to bank2 with no people in it (unsatisfiable)
```

```
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
    bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 0 & numCrossing(boat, cannibals) = 0.
```

Cplus2ASP Output:

Solving...

UNSATISFIABLE

Models : 0

No solution.

```
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & numOnBank(bank2, missionaries) = 0
    & numOnBank(bank2, cannibals) = 0 & loc(boat) = bank1;
  maxstep: numOnBank(bank2, missionaries) = 3 & numOnBank(bank2, cannibals) = 3.
```

Cplus2ASP Output:

Solving...

Solution: 1

0: loc(boat)=bank1 numOnBank(bank1, cannibals)=3 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=0 numOnBank(bank2, missionaries)=0
ACTIONS: cross(boat) numCrossing(boat, cannibals)=2 numCrossing(boat, missionaries)=0

1: loc(boat)=bank2 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=2 numOnBank(bank2, missionaries)=0
ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0

2: loc(boat)=bank1 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=1 numOnBank(bank2, missionaries)=0
ACTIONS: cross(boat) numCrossing(boat, cannibals)=2 numCrossing(boat, missionaries)=0

3: loc(boat)=bank2 numOnBank(bank1, cannibals)=0 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=3 numOnBank(bank2, missionaries)=0
ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0

4: loc(boat)=bank1 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=2 numOnBank(bank2, missionaries)=0
ACTIONS: cross(boat) numCrossing(boat, cannibals)=0 numCrossing(boat, missionaries)=2

5: loc(boat)=bank2 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=1 numOnBank(bank2, cannibals)=2 numOnBank(bank2, missionaries)=2
ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=1

6: loc(boat)=bank1 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=2 numOnBank(bank2, cannibals)=1 numOnBank(bank2, missionaries)=1
ACTIONS: cross(boat) numCrossing(boat, cannibals)=0 numCrossing(boat, missionaries)=2

7: loc(boat)=bank2 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=1 numOnBank(bank2, missionaries)=3
ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0

8: loc(boat)=bank1 numOnBank(bank1, cannibals)=3 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=0 numOnBank(bank2, missionaries)=3
ACTIONS: cross(boat) numCrossing(boat, cannibals)=2 numCrossing(boat, missionaries)=0

9: loc(boat)=bank2 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=2 numOnBank(bank2, missionaries)=3
ACTIONS: cross(boat) numCrossing(boat, cannibals)=0 numCrossing(boat, missionaries)=1

10: loc(boat)=bank1 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=1 numOnBank(bank2, cannibals)=2 numOnBank(bank2, missionaries)=2
ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=1

11: loc(boat)=bank2 numOnBank(bank1, cannibals)=0 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=3 numOnBank(bank2, missionaries)=3

SATISFIABLE

The LLM chooses not revise anything, as it marks “[UNCHANGED]” for the \mathcal{BC}^+ signature and rules, main query, and sample queries.

```
% PROGRAM CHANGED? [UNCHANGED]
% BC+ signature
:- sorts
  vessel;
  location;
  group;
  integer.

:- objects
  boat :: vessel;
  bank1, bank2 :: location;
  missionaries, cannibals :: group;
  0..10 :: integer.

:- variables
  V, V1, V2, V3, V4 :: vessel;
  L, L1, L2, L3, L4 :: location;
  G, G1, G2, G3, G4 :: group;
  N, N1, N2, N3, N4 :: integer;
  Bool, Bool1, Bool2, Bool3, Bool4 :: boolean.

:- constants
  cross(vessel) :: exogenousAction;
  numCrossing(vessel, group) :: attribute(integer) of cross(vessel);
  loc(vessel) :: inertialFluent(location);
  numOnBank(location, group) :: additiveFluent(integer).

% Constraints/rules

% 1. Crossing the river in the vessel causes the location of the vessel to change to the opposite bank.
cross(V) causes loc(V) = bank2 if loc(V) = bank1.
cross(V) causes loc(V) = bank1 if loc(V) = bank2.

% 2. Crossing the river in the vessel causes the number of people of each group on the departure bank to
% decrease by the number of people of that group crossing in the vessel.
cross(V) decrements numOnBank(L, G) by N if numCrossing(V, G) = N & loc(V) = L.

% 3. Crossing the river in the vessel causes the number of people of each group on the arrival bank to increase
% by the number of people of that group crossing in the vessel.
% 3.a If the vessel is at bank1, crossing increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V)
% = bank1.
cross(V) increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V) = bank1.

% 3.b If the vessel is at bank2, crossing increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V)
% = bank2.
cross(V) increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V) = bank2.

% 4. The vessel cannot cross the river with zero people on it.
nonexecutable cross(V) if N1 + N2 = 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 5. The vessel cannot carry more than two people at once.
nonexecutable cross(V) if N1 + N2 > 2 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 6. The number of people of each group crossing in the vessel cannot exceed the number of people of that group
% on the bank where the vessel is located.
nonexecutable cross(V) if N1 > N2 & N1 = numCrossing(V, G) & N2 = numOnBank(L, G) & loc(V) = L.
```

```

% 7. The number of people of each group crossing in the vessel cannot be negative.
nonexecutable cross(V) if N < 0 & N = numCrossing(V, G).

% 8. After crossing, on either bank, the number of cannibals cannot exceed the number of missionaries unless the
   number of missionaries on that bank is zero.
impossible N1 > N2 & N2 > 0 & N1 = numOnBank(L, cannibals) & N2 = numOnBank(L, missionaries).

% 9. During crossing, the number of cannibals crossing in the vessel cannot exceed the number of missionaries
   crossing in the vessel unless the number of missionaries crossing is zero.
nonexecutable cross(V) if N2 > N1 & N1 > 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 10. The number of people of each group on either bank cannot be negative or exceed three.
impossible N < 0 & N = numOnBank(L, G).
impossible N > 3 & N = numOnBank(L, G).

% MAIN QUERY CHANGED? [UNCHANGED]
% Three missionaries and three cannibals are at location bank1. The boat is at location bank1. Find a plan so
   that all missionaries and all cannibals are at location bank2.
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & numOnBank(bank2, missionaries) = 0
     & numOnBank(bank2, cannibals) = 0 & loc(boat) = bank1;
  maxstep: numOnBank(bank2, missionaries) = 3 & numOnBank(bank2, cannibals) = 3.

% SAMPLE QUERIES CHANGED? [UNCHANGED]
% Query 1: Have one missionary and one cannibal cross from bank1 to bank2 in the boat (satisfiable)
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
      bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 1 & numCrossing(boat, cannibals) = 1.

% Query 2: Try to have two missionaries and one cannibal cross from bank1 to bank2 in the boat (unsatisfiable)
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
      bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 2 & numCrossing(boat, cannibals) = 1.

% Query 3: Have one cannibal cross from bank1 to bank2 alone in the boat (satisfiable)
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
      bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 0 & numCrossing(boat, cannibals) = 1.

% Query 4: Try to have one missionary cross from bank1 to bank2 alone in the boat (unsatisfiable)
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
      bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 1 & numCrossing(boat, cannibals) = 0.

% Query 5: Try to have the boat cross from bank1 to bank2 with no people in it (unsatisfiable)
:- query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & loc(boat) = bank1 & numOnBank(
      bank2, missionaries) = 0 & numOnBank(bank2, cannibals) = 0;
  0: cross(boat) & numCrossing(boat, missionaries) = 0 & numCrossing(boat, cannibals) = 0.

```

The final correct program is generated as follows.

```

:- sorts
vessel;
location;
group;
integer.

```

```

:- objects
  boat :: vessel;
  bank1, bank2 :: location;
  missionaries, cannibals :: group;
  0..10 :: integer.

:- variables
  V, V1, V2, V3, V4 :: vessel;
  L, L1, L2, L3, L4 :: location;
  G, G1, G2, G3, G4 :: group;
  N, N1, N2, N3, N4 :: integer;
  Bool, Bool1, Bool2, Bool3, Bool4 :: boolean.

:- constants
  cross(vessel) :: exogenousAction;
  numCrossing(vessel, group) :: attribute(integer) of cross(vessel);
  loc(vessel) :: inertialFluent(location);
  numOnBank(location, group) :: additiveFluent(integer).

% Constraints/rules

% 1. Crossing the river in the vessel causes the location of the vessel to change to the opposite bank.
cross(V) causes loc(V) = bank2 if loc(V) = bank1.
cross(V) causes loc(V) = bank1 if loc(V) = bank2.

% 2. Crossing the river in the vessel causes the number of people of each group on the departure bank to
%     decrease by the number of people of that group crossing in the vessel.
cross(V) decrements numOnBank(L, G) by N if numCrossing(V, G) = N & loc(V) = L.

% 3. Crossing the river in the vessel causes the number of people of each group on the arrival bank to increase
%     by the number of people of that group crossing in the vessel.
% 3.a If the vessel is at bank1, crossing increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V)
%     = bank1.
cross(V) increments numOnBank(bank2, G) by N if numCrossing(V, G) = N & loc(V) = bank1.

% 3.b If the vessel is at bank2, crossing increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V)
%     = bank2.
cross(V) increments numOnBank(bank1, G) by N if numCrossing(V, G) = N & loc(V) = bank2.

% 4. The vessel cannot cross the river with zero people on it.
nonexecutable cross(V) if N1 + N2 = 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 5. The vessel cannot carry more than two people at once.
nonexecutable cross(V) if N1 + N2 > 2 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

% 6. The number of people of each group crossing in the vessel cannot exceed the number of people of that group
%     on the bank where the vessel is located.
nonexecutable cross(V) if N1 > N2 & N1 = numCrossing(V, G) & N2 = numOnBank(L, G) & loc(V) = L.

% 7. The number of people of each group crossing in the vessel cannot be negative.
nonexecutable cross(V) if N < 0 & N = numCrossing(V, G).

% 8. After crossing, on either bank, the number of cannibals cannot exceed the number of missionaries unless the
%     number of missionaries on that bank is zero.
impossible N1 > N2 & N2 > 0 & N1 = numOnBank(L, cannibals) & N2 = numOnBank(L, missionaries).

% 9. During crossing, the number of cannibals crossing in the vessel cannot exceed the number of missionaries
%     crossing in the vessel unless the number of missionaries crossing is zero.
nonexecutable cross(V) if N2 > N1 & N1 > 0 & N1 = numCrossing(V, missionaries) & N2 = numCrossing(V, cannibals).

```

% 10. The number of people of each group on either bank cannot be negative or exceed three.
impossible $N < 0$ & $N = \text{numOnBank}(L, G)$.
impossible $N > 3$ & $N = \text{numOnBank}(L, G)$.

% Three missionaries and three cannibals are at location bank1. The boat is at location bank1. Find a plan so
that all missionaries and all cannibals are at location bank2.

```
:= query
  0: numOnBank(bank1, missionaries) = 3 & numOnBank(bank1, cannibals) = 3 & numOnBank(bank2, missionaries) = 0
    & numOnBank(bank2, cannibals) = 0 & loc(boat) = bank1;
  maxstep: numOnBank(bank2, missionaries) = 3 & numOnBank(bank2, cannibals) = 3.
```

A correct plan is generated as output.

Solving...

Solution: 1

0: loc(boat)=bank1 numOnBank(bank1, cannibals)=3 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=0
numOnBank(bank2, missionaries)=0

ACTIONS: cross(boat) numCrossing(boat, cannibals)=2 numCrossing(boat, missionaries)=0

1: loc(boat)=bank2 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=2
numOnBank(bank2, missionaries)=0

ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0

2: loc(boat)=bank1 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=1
numOnBank(bank2, missionaries)=0

ACTIONS: cross(boat) numCrossing(boat, cannibals)=2 numCrossing(boat, missionaries)=0

3: loc(boat)=bank2 numOnBank(bank1, cannibals)=0 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=3
numOnBank(bank2, missionaries)=0

ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0

4: loc(boat)=bank1 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=3 numOnBank(bank2, cannibals)=2
numOnBank(bank2, missionaries)=0

ACTIONS: cross(boat) numCrossing(boat, cannibals)=0 numCrossing(boat, missionaries)=2

5: loc(boat)=bank2 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=1 numOnBank(bank2, cannibals)=2
numOnBank(bank2, missionaries)=2

ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=1

6: loc(boat)=bank1 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=2 numOnBank(bank2, cannibals)=1
numOnBank(bank2, missionaries)=1

ACTIONS: cross(boat) numCrossing(boat, cannibals)=0 numCrossing(boat, missionaries)=2

7: loc(boat)=bank2 numOnBank(bank1, cannibals)=2 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=1
numOnBank(bank2, missionaries)=3

ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=0

8: loc(boat)=bank1 numOnBank(bank1, cannibals)=3 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=0
numOnBank(bank2, missionaries)=3

ACTIONS: cross(boat) numCrossing(boat, cannibals)=2 numCrossing(boat, missionaries)=0

9: loc(boat)=bank2 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=2

```

numOnBank(bank2, missionaries)=3

ACTIONS: cross(boat) numCrossing(boat, cannibals)=0 numCrossing(boat, missionaries)=1

10: loc(boat)=bank1 numOnBank(bank1, cannibals)=1 numOnBank(bank1, missionaries)=1 numOnBank(bank2, cannibals)=2 numOnBank(bank2, missionaries)=2

ACTIONS: cross(boat) numCrossing(boat, cannibals)=1 numCrossing(boat, missionaries)=1

11: loc(boat)=bank2 numOnBank(bank1, cannibals)=0 numOnBank(bank1, missionaries)=0 numOnBank(bank2, cannibals)=3 numOnBank(bank2, missionaries)=3

SATISFIABLE
Models : 1+
Calls : 1
Time : 1.129s (Solving: 0.18s 1st Model: 0.18s Unsat: 0.00s)
CPU Time : 1.129s

```

B Complete Prompts

Capital words encapsulated like “<WORDS>” represent problem specific text, such as problem inputs, or outputs from previously generated steps. Lowercase words encapsulated like “<words>” are not problem specific and are used to reduce verbosity. The prompt templates and exact inputs to the LLM are found in the code repository.

A *BC+* description is put in all prompts, except the prompt for knowledge generation, since the knowledge generated is in natural language instead of any formal representation.

Action language BC+ consists of (1) a signature, which is a finite set of propositional atoms, and (2) causal laws. An atom is true, false, or of the form $c=v$, where c is a constant, and v is an element in its domain. Constants c are divided into two groups: fluent constants and action constants. A formula is a propositional combination of atoms. A fluent formula is a formula such that all constants occurring in it are fluent constants. An action formula is a formula that contains at least one action constant and no fluent constants.

For example, a fluent constant "loc" which represents the location of some object, may have a value from {11, 12, 13}. To express the location is 12, one would write:
`loc = 12.`

Similarly, an action constant "drive" representing driving a car, may have a value from {true, false} (though in general actions can be non-boolean). To represent this, one could write:
`drive = true.`

For boolean constants c , c is shorthand for $c=true$, and $\sim c$ is shorthand for $c=false$. Thus "drive = true" can be represented simply as "drive". The standard logical operators are: " \neg " for negation, " $\&$ " for conjunction, " \mid " for disjunction, and " \rightarrow " for implication.

Numeric values may combine with unary operators " \neg " (negative), "abs" (absolute value, e.g., `abs(n)`), and binary operators "+" (addition), " \neg " (subtraction), "*" (multiplication), " $\//$ " (floor division), and "mod" (modulus, e.g., $n \bmod m$). Comparison is done with " $=$ " (equality), " \neq " (inequality), " $<$ " (less than), " $>$ " (greater than), " \leq " (less than or equal), and " \geq " (greater than or equal).

An action description D is a set of causal laws that define a transition system, which can be represented as a directed graph. In this graph, the vertices correspond to states of the world, while the edges signify transitions between these states.

[CAUSAL LAWS: BASIC FORM]

In BC+, there are three types of basic laws: (1) static law, (2) action dynamic law, and (3) fluent dynamic law:

STATIC LAW:

`F if G,`

where F and G are fluent formulas. This expresses that F is true if the formula G is true.

ACTION DYNAMIC LAW:

F if G,

where F is an action formula and G is a formula. This expresses that action formula F is true if G is true.

FLUENT DYNAMIC LAW:

F if G after H,

where F and G are fluent formulas, and H is a formula. This expresses that F is true if G is true after H is true.

[CAUSAL LAWS: SOME SHORTHAND ABBREVIATIONS]

BC+ allows several kinds of shorthand notation for the basic causal laws. These make it convenient to directly express commonsense knowledge.

CAUSES:

a causes F if H,

where a is an action, F is a fluent formula, H is a formula, is shorthand for the fluent dynamic law:

F after a & H.

This rule intuitively expresses that an action a causes some effect F if condition H is true.

IMPOSSIBLE:

impossible F,

where F is a fluent formula, is shorthand for the static law:

false if F.

This rule intuitively expresses that formula F should be false in every state.

NONEXECUTABLE:

nonexecutable a₁ & ... & a_k if G,

where a₁ & ... & a_k are action constants and G is a formula, is shorthand for the fluent dynamic law:

false if true after a₁ & ... & a_k & G.

This rule intuitively expresses that actions a₁ & ... & a_k all being true is not possible if G is true.

DEFAULT:

default c=v if F,

where F is a fluent formula, stands for a static causal law.

This intuitively expresses that by default c=v holds when F is true in the state. If there is evidence to the contrary, then c=v may not hold.

default c=v if F after G,

where F is a fluent formula, and G is a formula, and stands for a fluent dynamic law which intuitively expresses that by default c=v holds when F is true in the state and F was true in the previous state. If there is evidence to the contrary, then c=v may not hold.

ALWAYS:

always F,

where F is a formula which can contain both fluent and action constants, is shorthand for the fluent dynamic law :

false if true after $\sim F$.

This rule intuitively expresses that every transition must satisfy F.

[SORT/OBJECT/VARIABLE/CONSTANT DECLARATION IMPLEMENTATION]

Action language BC+ is implemented in a program as follows. A BC+ program consists of a sort declaration, object declaration, variable declaration, constant declaration, along with a set of causal laws.

SORT DECLARATION:

A sort is a named set of elements which is used to specify the domain of each constant and variable. First the sort is declared using a sort declaration statement and, later, is defined by adding objects to it in an object declaration statement.

Example:

```
:-- sorts
    int;
    box.
```

This declares the sorts: int, and box.

A sort can also be a supersort, that is, a sort s can automatically include the objects within another sort s2, denoted $s \gg s_2$.

```
:-- sorts
    package >> box.
```

This declares the sort package and box, where package is a supersort and includes the sort box.

OBJECT DECLARATION:

An object is a value in a sort which a constant can take. It is also used in parameter lists to construct nested objects and sets of constants.

Example:

```
:-- objects
    1..3 :: int;
    o(int, int) :: box.
```

This declares 1, 2, and 3 as objects within int, and the objects:

$o(1, 1)$, $o(1, 2)$, $o(1, 3)$,
 $o(2, 1)$, $o(2, 2)$, $o(2, 3)$,
 $o(3, 1)$, $o(3, 2)$, and $o(3, 3)$,
as values within box.

Consider the following sort and object declaration:

```
:-- sorts
    package >> box.

:-- objects
    p1, p2 :: package;
    b1, b2, b3 :: box.
```

Since package is a supersort of box, there are 5 objects that are of sort package: p1, p2, b1, b2, and b3. There are only 3 objects of sort box: b1, b2 and b3.

VARIABLE DECLARATION:

A variable is a placeholder symbol which will be replaced with each object in its domain during grounding.

Example:

```
:- variables
  I, I1, I2 :: int;
  B, B1, B2 :: box.
```

This declares the variables I, I1, and I2 over the objects within int, and variables B, B1, and B2 range over the objects within box.

Variables are denoted as capital letters which are replaced with all possible values in the domain of their associated sort in a process called grounding.

CONSTANT DECLARATION:

Similar to object symbols, constants are defined within a constant declaration statement and have a base identifier, an optional list of parameter sort, and a sort which makes up the constant's domain. In addition, they have a constant declaration type, such as "exogenousAction" or "inertialFluent". Inertial fluents are common, and used for constants whose value persists through time unless affected otherwise.

Example:

```
:- constants
  p(int) :: inertialFluent(boolean);
  move(box) :: exogenousAction.
```

This declares the inertial fluent p which takes an int argument and is a boolean value, and the action move, which takes a box argument.

[CAUSAL LAWS BASIC FORM IMPLEMENTATION]

STATIC LAW EXAMPLE:

F if G,

where F and G are fluent formulas, is written for example:

% The location of a person is the same as the car if the person is in the car.
loc(person)=L if inCar & loc(car)=L.

ACTION DYNAMIC LAW EXAMPLE:

F if G,

where F is an action formula and G is a formula, is written for example:

% If A is assigned to location Loc, and it is ready, then moveTo(A) is true.
moveTo(A) if assigned(A) = Loc & ready(A),
where moveTo(A) is an action, and assigned(A) and ready(A) are fluent constants. Here moveTo is a boolean action constant.

FLUENT DYNAMIC LAW EXAMPLE:

F if G after H,

where F and G are fluent formulas, and H is a formula, is written for example:

% If you enter the car after the car is unlocked, then you are in the car.
in(car) if unlocked(car) after enter(car).

[CAUSAL LAW SHORTHAND ABBREVIATIONS IMPLEMENTATION]

CAUSES EXAMPLE:

An expression of the form
a causes F if H,

where a is an action, F is a fluent formula, H is a formula, is written for example:
% Driving the car to a location causes the car to be at that location of the car has gas.
drive(car, loc) causes location(car) = loc if hasGas(car)

IMPOSSIBLE EXAMPLE:

An expression of the form

impossible F,

where F is a fluent formula, is written for example:

% The cat and mouse cannot be at the same location
impossible loc(cat)=Loc1 & loc(mouse) = Loc2 & Loc1=Loc2.

Recall that F in "impossible F" must be a fluent formula, meaning that F cannot contain action constants, including attributes.

The following two examples are incorrect, where drive(car) is an action, and numPassengers(car) and numDrivers(car) are attributes:

% The car cannot be driven if it has no fuel.
impossible drive(car) & fuel(car) = 0.

% The combined number of passengers and drivers cannot exceed 5.
impossible N1 + N2 > 5 & N1 = numPassengers(car) & N2 = numDrivers(car).

These are wrong because drive(car) is an action constant, and the attributes numPassengers and numDrivers are action constants, and they are in an impossible constraint.

NONEXECUTABLE EXAMPLE:

An expression of the form:

nonexecutable a1 & a2 & ... & an if G,

where a_1 & ... & a_k are action constants and G is a formula, is written for example:

% It is not permissible to drive a bus and a car if they are driving to the same location.
nonexecutable driveCar(Loc1) & driveBus(Loc2) if Loc1=Loc2.

% It is not permissible to lift an object if it is heavy.
nonexecutable lift(object) if heavy(object).

On the other hand, the following:

nonexecutable lifted(object) if heavy(object),

is not allowed when lifted(object) is a fluent and not an action.

DEFAULT EXAMPLES:

An expression of the form

default c=v if F,

where F is a fluent formula, is written for example as:

% By default, the power is on if the switch is turned.
default powerOn if switchTurned.

An expression of the form

default c=v if F after G,

where F is a fluent formula, and G is a formula, is written for example as:

% The ferry will be on the right bank if it has gas after it was on the left bank, and no evidence suggests otherwise.
default ferry(right) if hasGas after ferry(left)

ALWAYS EXAMPLE:

```
always F,
```

where F is a formula which can contain both fluent and action constants, is written for example:

```
% the number departing from a location is always less than or equal to the number at the location  
always departing(Loc) <= numAt(Loc).
```

where departing/1 is an action constant and numAt/1 is a fluent.

Note: There should not be an "if" in a "always" rule, instead use "&".

Keep in mind that "always F" should be used for conditions which should be true for every transition.

[EXTENDED CONSTANT DECLARATION]

We can expand constant declaration to allow for attributes, additive fluents and additive actions.

ATTRIBUTES:

Attributes are types of action constants which are tied to actions, and given additional information about an action. For instance:

```
:- constants  
    move(block) :: exogenousAction;  
    destination(block) :: attribute(location) of move(block).
```

This declaration tells us that destination(block) is an attribute of the action of moving a block, which has value of location, representing the block's destination.

Important: Attributes should have as their first arguments, the same arguments (and in the same order) as the action which they are an attribute of.

For example, any attribute of the action "fly(airplane,location)" should have the form "attributeName(airplane, location,...)", after which the action arguments "airplane" and "location", can other arguments be listed.

An attribute which represents the number of passengers on board of a nationality can be written as:

```
onBoard(airplane,location,nationality) :: attribute(integer) of fly(airplane,location).
```

Where the following would be incorrect:

Incorrect: passengers(nationality) :: attribute(integer) of fly(airplane,location),
since passengers should instead have the arguments "airplane" and "location" before "nationality".

Important: If an attribute is ever declared, then the action it is an attribute of must have at least one argument.

For example, the above declaration would be incorrect if it was the following:

```
:- constants  
    move :: exogenousAction;  
    destination :: attribute(location) of move.
```

Here, since destination is an attribute of move, but move has no argument, this is wrong.

ADDITIVE FLUENTS AND ADDITIVE ACTIONS:

There are special constants additive fluents and additive actions, which are useful for modeling the effects of concurrent actions.

An additive fluent is a fluent (tied to states) with numerical values such that the effect of several concurrently executed actions on it can be computed by adding the effects of the individual actions.

An additive action constant (tied to a transition) is similar except that it can handle numeric values of concurrent actions during transitions.

With a buy and sell action, and numExchanged attribute of exchange, additive fluents and additive actions can be declared as follows:

```
:- constants  
    buy(item) :: exogenousAction;  
    sell(item) :: exogenousAction;  
    exchange(item) :: exogenousAction;
```

```

numExchanged(item) :: attribute(integer) of exchange(item);
inStock(item) :: additiveFluent(integer);
netChange(item) :: additiveAction(integer).

```

inStock(item) represents the the number of an item in stock, which can be affected by both buy(item) and sell(item). netChange(item) represents the net number of an item which is exchanged or bought/sold during a given transition or event.

[EXTENDED CAUSAL LAW SHORTHAND ABBREVIATIONS: INCREMENT/DECREMENT LAWS IMPLEMENTATION]

INCREMENT/DECREMENT LAWS:

Additive fluents can only be updated in increment/decrement laws, which are expressed as:

a increments c by v if G,

where a is an action, c is an additive fluent, v is an integer, and G is a fluent, is used to express that some additive fluent (c), should increase by some value v if G is true and action a happens.

Similarly:

a decrements c by v if G,

is used to decrease the additive constant c by value v if G is true and action a happens.

Examples:

```

% buy(coin) increments has(coin) by N if howMany(coin,N).
% buy(coin) decrements has(money) by N*M if howMany(coin,N) & price(coin)=M.

```

Additive actions similarly are only updated in increment/decrement statements:

```

a increments c by v if G.
a decrements c by v if G.

```

where c is an additive action.

While additive fluents are used to describe state information, additive actions are about events/transitions.

For example, suppose multiple actions can occur at once for a domain involving buying and selling, and the additive action netExchanged is declared:

```

% buy(coin) increments netExchanged(coin) by N if howMany(coin,N).
% sell(coin) decrements netExchanged(coin) by N if howMany(coin,N).

```

This is useful for tracking numerical values during a transition, which may be used, e.g., if there should never be a negative amount of net coins exchanged at a given time, then netExchanged(coin) can be used to construct such a law.

In general, if something causes something to increase/decrease, then we use increment/decrement. Updating the value of an additive fluent or action can only be done in an increment or decrement law (e.g., "a causes c =2" is not allowed if "c" represents an additive fluent/action).

[ARITHMETIC WITH ATTRIBUTES AND ADDITIVE CONSTANTS]

Attributes and additive actions should not occur in the operands of arithmetics such as addition, subtraction, multiplication, floor division, or modulus operators. They must be set to variables, and the variables must be used as the operands.

The following two rules are incorrect, where weight(truck,...) is an attribute:

```

% A truck cannot be driven if the combined weight of the cargo (C), passengers (P), and fuel (F), is greater than the truck's capacity.

```

```

nonexecutable drive(truck, P, C) if weight(truck, C) + weight(truck, P) + weight(truck, F) > capacity(truck).

```

```

% Driving the truck causes the weight of the truck's fuel to decrease by 15.

```

```

drive(truck, P, C) causes weight(truck, fuel) = weight(truck, fuel) - 15.

```

These are incorrect because the attributes weight(truck,...) are directly used in arithmetic.

Instead, these rules should be written as follows:

```

% A truck cannot be driven if the combined weight of the cargo (C), passengers (P), and fuel (F), is greater than the truck's capacity.

```

```
nonexecutable drive(truck, P, C) if N1 + N2 + N3 > capacity(truck) & weight(truck, C) = N1 & weight(truck, P) = N2 & weight(truck, F) = N3.
```

% Driving the truck causes the weight of the truck's fuel to decrease by 15.
`drive(truck, P, C)` causes `weight(truck, fuel) = N1 - 15` if `N1 = weight(truck, fuel)` & `N2`.

[RESTRICTION OF NESTED CONSTANTS]

A constant cannot appear as an argument of another constant. Instead, one can introduce a variable in place of a nested constant.

For example, the following is wrong since the (boolean) constant `roadBlock(...)`, contains the nested constant `loc(C)`:
`nonexecutable move(C) if roadBlock(loc(C)).`

Instead, `loc(C)` should be set to a variable, like the following:
`nonexecutable move(C) if roadBlock(L) & L = loc(C).`

[VARIABLE QUANTIFICATION]

When using "always F", make sure to quantify over variables if needed.

For example, say we want to express that the combined weight of passengers and the truck is less than the capacity of the truck. The following would be incorrect:

```
always weight(passengers) = N1 & weight(truck) = N2 & capacity(truck) = N3 & N1 + N2 < N3.
```

This is because of a quantification issue. This rule informally reads as "For all values N1, N2, and N3, the weight of the passengers is N1, the weight of the truck is N2, the capacity of the truck is N3, and $N1 + N2 < N3$." Since this will be grounded for every possible value that N1 can take it, there will be contradiction. Furthermore, the condition " $N1 + N2 < N3$ " will obviously not hold over all possible values of the variables N1, N2, and N3.

To correctly write this, first set the variables in the antecedent of an implication:

```
always (weight(passengers) = N1 & weight(truck) = N2 & capacity(truck) = N3) → N1 + N2 < N3.
```

This informally reads as "If the weight of the passengers is N1, the weight of the truck is N2, and the capacity of the truck is N3, then $N1 + N2 < N3$ ", and aligns with what we want to express.

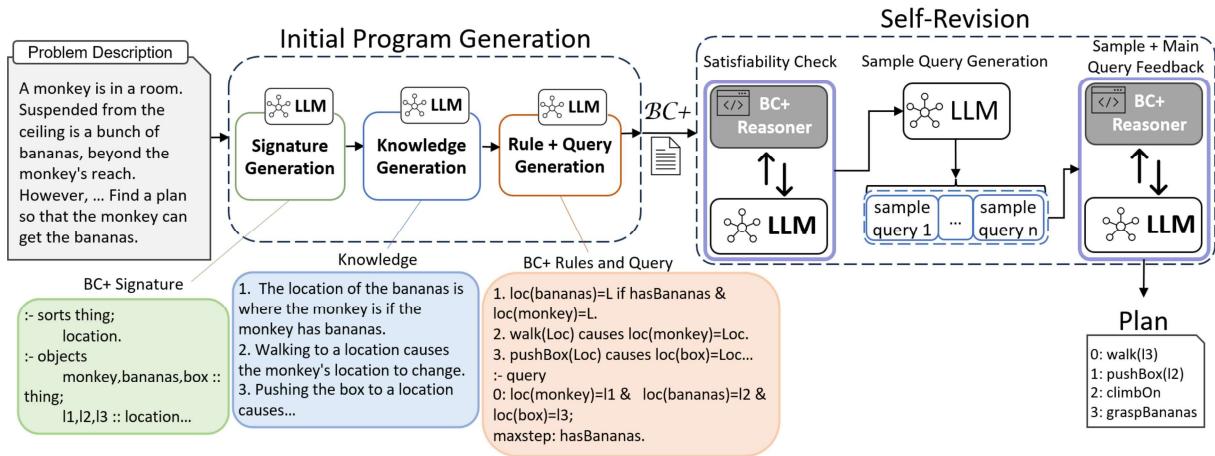


Figure 2: LLM+AL

B.1 BC+ Signature Generation

This prompt is used in the Signature Generation portion of Initial Program Generation, as shown in Figure 2.

```
<conceptual description of bc+>
```

[INSTRUCTIONS]

Given a hint, problem description, and knowledge about a domain, produce the relevant fluent and action constants and the BC+ signature. Do not use the same names for constants, even if they are of a different type or have different arguments. For example, on(block) and on(block,table) cannot both be declared.

<few-shot examples>

[Problem]

<DOMAIN>

As shown in the examples, write the actions, constants, and BC+ signature. Make sure to include enough variables for each sort, which will be needed later to write rules (5 variables each should be enough, for example, N, N1, N2, N3, N4 for integer, etc.).

Encapsulate your answer in 3 backticks like the following:

Actions:

<ACTIONS>

Constants:

<CONSTANTS AND THEIR READING>

BC+ Signature:

<SIGNATURE>

B.2 Knowledge Generation

This prompt is used in the Knowledge Generation portion of Initial Program Generation, as shown in Figure 2.

Given a description of a problem, write knowledge about the domain that we would expect to represent in an action language BC+ program. These should represent logical constraints and cause and effect in the problem.

For example, a task about moving objects would involve some knowledge about what is required to move, what the effect of moving an object does, such as the location of the object changing or its previous location now being empty. These restrictions and effects are dependent on the context of the problem. Here we list some example problems and extract knowledge.

<few-shot examples>

[Problem]

Description:

<PROBLEM DESCRIPTION> Provide the relevant knowledge. Only use the terminology in the hint/description to write the knowledge.

Hint:

<signature description>

BC+ description

<ACTIONS AND CONSTANTS>

<BC+ Signature>

Generate the the knowledge relevant to this problem, ONLY using natural langauge. Do not mention any of the program within this knowledge, though the signature should be considered when writing the knowledge.

Encapsulate it in three backticks like so:

% generated knowledge

% <knowledge #1>

% <knowledge #2>

B.3 Rule Generation

This prompt is used in the Rule and Query Generation portion of Initial Program Generation, as shown in Figure 2.

```
<conceptual description of bc+>

QUERIES:
Queries have three components: an optional label, an optional maximum step which lists formulas which should be true in the last step of the plan, and a set of constraint formulas to apply, each parametrized with the step at which they should be applied.
For example the following query represents an initial state and an action at step 0, and a set of constraints which should be met at the end of the plan.

:- query
    label :: test_query;
    0: loc(b1)=b2 & loc(b2)=b3 & loc(b3)=a & loc(b4)=b & loc(b5)=c;
    0: move(b1, b5);
    maxstep: loc(b5,c) & loc(b4,b5) & loc(b3,b4) & loc(b2,b3) & loc(b1,b2).

<few-show examples>

[INSTRUCTIONS]

Based on the problem description, previously generated BC+ signature and natural language rules/constraints to represent in BC+, generate the rules/constraints in BC+ along with the query.

[PROBLEM]

Description:
<PROBLEM DESCRIPTION>

Hint:
<HINT>

BC+ description:
<ACTIONS AND CONSTANTS>

<DOMAIN>

Rules/constraints to represent in BC+:
<CONSTRAINTS>

Query (in natural language):
<QUERY>

Do not alter or change the BC+ signature in any way, only present the constraints/rules (numbered), and query together in the following format, encapsulated in three backticks:

```
% constraints/rules
% 1. <comment for rule 1>
<BC+ rule 1>

% 2. <comment for rule 2>
<BC+ rule 1>
...

% query
< BC+ query>

```

```

B.4 Satisfiability Check

This prompt is used by the LLM in the Satisfiability Check portion of Self-Revision, as shown in Figure 2.

```

<conceptual description of bc+>

[INSTRUCTIONS]
Consider the problem and BC+ program, which when run with a basic query to check satisfiability (independent of the main query), fails.

[Problem]
<PROBLEM DESCRIPTION>

BC+ Program:
<BC+ PROGRAM>

Main query:
<QUERY>
<BC+ QUERY>

Feedback:
<FEEDBACK>

Based on the feedback, revise the program (signature and/or rules), and possibly the main query if needed to match a new signature. The format should be like the following, with everything together, encapsulated in 3 backticks:

```% BC+ signature
<Enter BC+ signature>

% Generated constraints
<Enter BC+ rules/constraints>

% Main query
<Enter the main query>```

```

## B.5 Sample Query Generation

This prompt is used by the LLM in the Sample Query Generation portion of Self-Revision, as shown in Figure 2.

```

<conceptual description of bc+>

QUERIES:
Queries have three components: an optional label, an optional maximum step which list formulas which should be true in the last step of the plan, and a set of constraint formulas to apply, each parametrized with the step at which they should be applied.
For example the following query represents an initial state and an action at step 0, and a set of constraints which should be met at the end of the plan.

:- query
 label :: test_query;
 0: loc(b1)=b2 & loc(b2)=b3 & loc(b3)=a & loc(b4)=b & loc(b5)=c;
 0: move(b1, b5);
 maxstep: loc(b5,c) & loc(b4,b5) & loc(b3,b4) & loc(b2,b3) & loc(b1,b2).

[INSTRUCTIONS]

Based on the problem description, and generated BC+ program. Generate some simple queries which can be used to make sure the program is working properly. For example, consider the following example.

<few-shot examples>

[PROBLEM]

Description:
<PROBLEM DESCRIPTION>

BC+ description:

```

```

<ACTIONS AND CONSTANTS>
<DOMAIN>
Rules/constraints to represent in BC+:
<CONSTRAINTS>
Main Query:
<QUERY>
<BC+ QUERY>

Based on the problem description, and generated BC+ program. Generate some simple queries which can be used to
make sure the program is working properly. Only generate a few, no more than 5. Append either (satisfiable)
or (unsatisfiable) to the end of the query in natural language, depending on whether the query should work
or not. Keep in mind, the starting states of the sample queries should be allowable based on the already
written constraints in the program (e.g., if a query comes back unsatisfiable, it shouldn't be because the
start state was unsatisfiable). Do so in the following format, encapsulated in 3 backticks:
```
% Query 1: <natural language query> (satisfiable/unsatisfiable)
<BC+ Query>

% Query 2: <natural language query> (satisfiable/unsatisfiable)
<BC+ Query>
...
```

```

## B.6 Sample and Main Query Feedback

This prompt is used by the LLM in the Sample and Main Query Feedback portion of Self-Revision, as shown in Figure 2.

```

<conceptual description of bc+>

QUERIES:
Queries have three components: an optional label, an optional maximum step which lists formulas which should be
true in the last step of the plan, and a set of constraint formulas to apply, each parametrized with the
step at which they should be applied.
For example the following query represents an initial state and an action at step 0, and a set of constraints
which should be met at the end of the plan.

:- query
 label :: test_query;
 0: loc(b1)=b2 & loc(b2)=b3 & loc(b3)=a & loc(b4)=b & loc(b5)=c;
 0: move(b1, b5);
 maxstep: loc(b5,c) & loc(b4,b5) & loc(b3,b4) & loc(b2,b3) & loc(b1,b2).

<few-shot examples>

[CHECKLIST]
The following is a checklist which should be considered when writing and revising a BC+ program.
1. In general, if some conjunction of fluents cannot be true, then we use "impossible" when writing the rules,
but if instead we want to assert some constant is caused to be false, then we use the negation (~). For
example, if it is impossible for an object to be on the table and under it, we might write "impossible
onTable(object) & underTable(object).", but express that no rain causes the ground to not be wet, we can
write "~groundWet if noRain".
2. "impossible F" can only be used with fluents and "&". Do not use attributes or action constants in laws of
the form "impossible F". "nonexecutable" is about actions not being permissible if certain fluents hold and
/or certain actions are performed.
3. Additive constants can only be updated in increment/decrement laws.
4. In "a causes G if H", "G", recall G should be a fluent formula. Therefore, it should not directly contain
action constants (including attributes). If one wants to set some constant to the value of an attribute,
then the fluent formula G should contain a variable representing the attribute, rather than the attribute
itself.

```

For example, if the attribute nextDestination(Loc) is not set to a variable first, then this will be a malformed law error, since there is an action constant in G:  
`drive(car) causes loc(car) = nextDestination(Loc) if loc(boat) = Loc.`

Instead, this should be written as:

`drive(car) causes loc(car) = Loc2 if Loc2 = nextDestination(Loc) & loc(boat) = Loc.`

This works because nextDestination(Loc) is first set to a variable.

5. Only use variables which are declared, if new variables are introduced, they must be declared.
6. In shorthand laws "default c=v if F" and "default c=v if F after G", "c" cannot be an additive constant, since this is can update the value of them, but they should only be updated in increment/decrement laws.
7. "always" shorthand abbreviations should not contain "if", but only conjunctions ("&").
8. In the case that a disjunction is used in the body of a rule, encapsulate it in parentheses, e.g., `(loc(city1) | loc(city2))`.
9. Do not nest constants in arguments of other constants. For example, do not use the following: nonexecutable move(C) if roadBlock(loc(C)). Instead, loc(C) should be set to a variable: nonexecutable move(C) if roadBlock(L) & L = loc(C).
10. Make sure that the query is exhaustive. For example, if the initial condition is that bus stop A has 10 people, and the goal condition is that the bus stop B has 10 people, it may need to be specified that bus stop B has 0 people in the initial condition, or the BC+ solver can just set the initial state to be the goal state.
11. Make sure to only use constants which are declared in the constant declaration. If new constans are used, then they must be added in the declaration.
12. Make sure when writing rules, that variables representing constants are of the correct sort, i.e., the sort of the variable is the same as the sort of the value of the constant it is representing.

For example, if there is a constant "capacity(vehicle)" which is of type "inertialFluent(integer)", then only variables of sort "integer" should be used to represent the constant in rules.

13. Don't use simple, or rigid fluents in the constant declaration. Use "inertialFluent" instead.
14. Don't use non-specified fluents. For example, there should not be constant declarations like "country(location) :: country", since this will be recognized as a rigid fluent.
15. Keep in mind that "always F" should be used for conditions which should be true for every transition and shout not be used to express conditions for a state.

For example, consider a domain about trading coins that has the action constant numExchanged(coin), which is an attribute of some trading action, and the fluent constant amount(coin). numExchanged(coin) represents the net number of coins exchanged in a transition, from possibly multiple trade actions. amount(coin) represents the number of coins in a state (before and after trades).

If one wants to express that the net number of exchanged coins during an action is smaller than some limit for every trade, this can be expressed with:

`always netExchanged(coin) < limit(coin).`

However, if one wants to express that the amount of coins does not exceed some limit, this cannot be expressed with "always F". The following is incorrect:

`always amount(coin) <= limit(coin).`

This wrong because if a state does not have a transition to another state, then the condition "amount(coin) <= limit(coin)" can be false. For example, in a transition system with a single state, or the last state in a transition system.

Instead, it can be expressed as:

`impossible amount(coin) > limit(coin).`

This enforces the following is true for every state.

16. Attributes and additive actions should not occur in the operands of arithmetics such as addition, subtraction, multiplication, floor division, or modulus operators. They must be set to variables, and the variables must be used as the operands.

#### [INSTRUCTIONS]

Consider the problem and BC+ program, and simple sample queries which were ran to check the basic functionality of the program. The output of each sample query and the main query is presented. Check that the Cplus2ASP output makes sense.

[Problem]

<PROBLEM DESCRIPTION>

BC+ Program:  
 <BC+ PROGRAM>

Sample queries and outputs:  
 <FEEDBACK>

Main query and output:  
 % Original natural language query: <QUERY>  
 <FEEDBACK MAIN QUERY>

[INSTRUCTIONS (continued)]

- 1) [FEEDBACK OUTLINE]
 

Based on the feedback, if any of the following segments are incorrect, then mark them with "[CHANGED]", and revise them:

  - PROGRAM
  - MAIN QUERY
  - SAMPLE QUERIES

Otherwise, if a segment is correct, mark it with "[UNCHANGED]", and simply copy their contents.
- 2) [SAMPLE QUERY GUIDANCE]
 

Keep in mind, the outputs from running the sample queries is to help check that the program is working properly. In some cases, the sample queries themselves may be wrong (e.g., a syntax error), and can be re-written. The sample queries marked (satisfiable) should be correctly be satisfiable and the queries marked (unsatisfiable) should be unsatisfiable. There should not be apparent violations in the constraints, and the action preconditions/effects should enforced properly. If there are, then revise the main program. Overall, changes should be succinct, e.g., only change relevant parts while leaving the remainder alone.
- 3) [MAIN QUERY OUTPUT]
 

**IMPORTANT:** Check that each action in the final plan of the main query makes sense given the state before it, and that the resulting state correctly follows.

If something doesn't align with the common sense relative to the problem description, then update the program. Though less likely, if you suspect there is an issue with the main BC+ query, it may be changed, but it should have the original meaning of the main query as shown under "Main query and output", after "Original natural language query: ". The query itself should not contain actions, but have only the initial state and the final state (indicated with "maxstep").
- 4) [FORMAT]
 

The format should be like the following, encapsulated in 3 backticks:

```
```% PROGRAM CHANGED? [CHANGED/UNCHANGED]
% BC+ signature
<Enter BC+ signature>

% Generated constraints
<Enter BC+ rules/constraints>

% MAIN QUERY CHANGED? [CHANGED/UNCHANGED]
<Enter the main query>

% SAMPLE QUERIES CHANGED? [CHANGED/UNCHANGED]
<Enter the sample queries (do NOT include the Cplus2ASP outputs)>
```

```

## C Baseline Prompts

### C.1 Baseline LLM (CHATGPT-4, CLAUDE 3 OPUS, GEMINI 1.0 ULTRA, O1-PREVIEW)

The format of the prompt used is:

```
<PROBLEM DESCRIPTION>

<QUERY>
```

For example, for the River Cross domain:

There are four types of things: vessel, location, group, and integer.

Problem Description:

Three missionaries and three cannibals shall cross a river in a boat which carries at most two people at once.

The boat cannot cross the river with nothing on it. On either side, the cannibals may never outnumber the missionaries or the missionaries will be eaten. Also, the cannibals may never outnumber the missionaries on the boat or the missionaries will be eaten. The boat's capacity is 2.

boat is a vessel. bank1 and bank2 are locations. missionaries and cannibals are groups. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 are integers.

Three missionaries and three cannibals are at location bank1. The boat is at location bank1. Find a plan so that all missionaries and all cannibals are at location bank2.

## C.2 LLM + Code (CHATGPT-4+CODE)

Use the code interpreter to generate a Python program to solve the following problem. Revise the program as many times as needed until the program is correct.

<PROBLEM DESCRIPTION>

<QUERY>

## D LLM + AL Error Cases

The following contains all issues from LLM+AL after Self-Revision. Unless otherwise stated, all issues shown contribute 1 issue to the total number of issues.

### D.1 Signature

**Missing sort, object, variable, or constant in the declaration.**

Sudoku1 - the sort “int” needs objects instantiated from 0 to 8 (written “0..8”). This is required because arithmetic is done with variables of the sort “int”, so there must exist objects (representing integers) of “int”.

```
:- objects
 0..9 :: value0;
 1..9 :: value;
 0..8 :: row;
 0..8 :: column.
 % Missing: 0..8 :: int.
```

Sudoku (var2) - There are missing object declarations for the row, column and integer.

```
:- objects % Missing
 1..9 :: row; % Missing
 1..9 :: column; % Missing
 0..9 :: integer. % Missing
```

Sudoku (var2) - There is a missing sort declaration for integer.

```
:- sorts
 row;
 column;
 value;
 integer. % Missing
```

MCP #6 (not everyone can row) - There is a missing constant which tracks the location of rowers, rowersAt (basicGroup).

```
:- constants
 cross(vessel) :: exogenousAction;
 numCrossing(vessel, basicGroup) :: attribute(integer) of cross(vessel);
 rower(vessel, basicGroup) :: attribute(boolean) of cross(vessel);
```

```

powerAt(basicGroup) :: inertialFluent(location); % Missing
capacity(vessel) :: inertialFluent(integer);
boatLoc :: inertialFluent(location);
numAt(basicGroup, place) :: additiveFluent(integer).

```

MCP #9 (big cannibal and small missionary) - Variables L1, L2, N1, and N2 were used in rules, but never declared. Thus they are missing from the signature.

```

:- variables
M, M1, M2, M3 :: missionary;
C, C1, C2, C3 :: cannibal;
P, P1, P2 :: person;
Loc, Loc1, Loc2, NewLoc :: place;
N.m, N.c, N :: integer;
L1, L2, N1, N2 :: integer. % Missing

```

MCP #14 (the boat leaks with two people on it) - The action bail is missing from the constant declaration.

```

:- constants
cross(vessel) :: exogenousAction;
from(vessel) :: attribute(location) of cross(vessel);
to(vessel) :: attribute(location) of cross(vessel);
numOnBoard(vessel, group) :: attribute(integer) of cross(vessel);
loc(vessel) :: inertialFluent(location);
numOnBank(group, location) :: inertialFluent(integer);
numOnBoat(group) :: inertialFluent(integer);
capacity(vessel) :: inertialFluent(integer);
bail(vessel) :: exogenousAction. % Missing

```

MCP #17 (cannibals can become hungry) - A fluent constant which represents whether the cannibals are hungry is missing.

```

:- constants
loc(vessel) :: inertialFluent(location); % The location of the vessel.
capacity(vessel) :: inertialFluent(integer); % The capacity of the vessel.
cross(vessel) :: exogenousAction; % Crossing action.
numCrossing(vessel, group) :: attribute(integer) of cross(vessel); % Number from a group crossing in the vessel.
toLocation(vessel) :: attribute(location) of cross(vessel); % Destination location.
numAt(group, location) :: additiveFluent(integer); % Number of people of a group at a location.
hungry :: inertialFluent(boolean). % Missing

```

**Syntax Issues.** MCP # 19 (there are two sets of groups) - Variables are incorrectly used in the constant declaration. This is syntactically wrong, only sorts should be used as arguments. S, G, and L should be changed to set, group, and location.

```

:- constant
cross(S) :: exogenousAction;
dest(S) :: attribute(location) of cross(S);
numCrossing(S, G) :: attribute(integer) of cross(S);
boatLoc(S) :: inertialFluent(location);
numAt(S, G, L) :: additiveFluent(integer).

```

**Semantic Issues.** MCP # 8 (a very big cannibal must cross alone) - The following sort declaration incorrectly states that missionaries and cannibals are supersorts of each other. Rather, there should be semicolons after the first occurrence of missionary and cannibal.

```

:- sorts
integer;
missionary,
cannibal,
person >> missionary, cannibal;
vessel;
location.

```

MCP #9 (big cannibal and small missionary) - (1) The supersort statement is incorrect. person should be a supersort of missionary and cannibal, not the opposite. (2) vessel should not be a place, thus the supersort statement place >> location, vessel is wrong.

```

:- sorts
missionary >> person; % should be person >> missionary
cannibal >> person; % should be person >> cannibal
vessel;
location;
place >> location, vessel; % should be just place >> location
thing >> person, vessel;
integer.

```

MCP #5 (an oar on each bank) - The constants numMissionariesOn, numCannibalsOn, and numOarsOn, should be attributes rather than inertial fluents.

```

:- constants
cross(vessel) :: exogenousAction;
loc(vessel) :: inertialFluent(location);
capacity(vessel) :: inertialFluent(integer);
numMissionariesAt(location) :: inertialFluent(integer);
numCannibalsAt(location) :: inertialFluent(integer);
numOarsAt(location) :: inertialFluent(integer);
numMissionariesOn(vessel) :: inertialFluent(integer); % should be an attribute
numCannibalsOn(vessel) :: inertialFluent(integer); % should be an attribute
numOarsOn(vessel) :: inertialFluent(integer). % should be an attribute

```

## D.2 Rule

**Missing necessary rules.** Tower of Hanoi (3, var1) - The constraint which says that two disks cannot be at the same location is missing:

```
impossible loc(D) = L & loc(D1) = L & D \= D1.
```

Sudoku (var2) - The following state constraints are required. This is counted as 3 separate issues.

```

% No two cells in the same row have the same value
false if val(R, C) = V & val(R, C1) = V & C \= C1.

% No two cells in the same column have the same value
false if val(R, C) = V & val(R1, C) = V & R \= R1.

% No two cells in the same 3x3 block have the same value
false if val(R, C) = V & val(R1, C1) = V & (R \= R1 | C \= C1) &
(R // 3 = R1 // 3) & (C // 3 = C1 // 3).

```

MCP #6 (only one missionary and one cannibal can row) - There is a missing rule required to update the rorer locations.

```
cross(boat) causes rorerAt(BG) = L if boatLoc = L1 & rorer(boat, BG) & L \= L1.
```

MCP #6 (only one missionary and one cannibal can row) - There is a missing rule required which disallows the rorer to cross if it is not where the boat is.

```
nonexecutable cross(boat) if rorer(boat, BG) & rorerAt(BG) \= L & boatLoc = L.
```

MCP #8 (a very big cannibal must cross alone) - The are missing rules to represent that the cannibals cannot outnubmer the missionaries.

```

% disallow 3 cannibals and 2 missionaries
impossible loc(M) = Loc & loc(M1) = Loc & loc(M2) = Loc1 & M \= M1 &
loc(C) = Loc & loc(C1) = Loc & loc(C2) = Loc & C \= C1 & C \= C2 & C1 \= C2 &
Loc \= Loc1.

% disallow 3 cannibals and 1 missionary
impossible loc(M) = Loc & loc(M1) = Loc1 & loc(M2) = Loc1 & M1 \= M2 &
loc(C) = Loc & loc(C1) = Loc & loc(C2) = Loc & C \= C1 & C \= C2 & C1 \= C2 &
Loc \= Loc1.

% disallow 2 cannibals and 1 missionary

```

```
impossible loc(M) = Loc & loc(M1) = Loc1 & loc(M2) = Loc1 & M1\=M2 &
loc(C) = Loc & loc(C1) = Loc & loc(C2) = Loc1 & C\=C1 & C\=C2 &
Loc \= Loc1.
```

MCP # 14 (the boat leaks with two people on it) - The constraint which disallows crossing to your current location is missing.

```
nonexecutable cross(V) if to(V) = loc(V).
```

MCP # 14 (the boat leaks with two people on it) - The rule which ensures bailing is done when two people are on board is missing is missing.

```
bail(V) if numOnBoard(V, missionaries) = N1 & numOnBoard(V, cannibals) = N2 & N1 + N2 = 2.
```

MCP #17 (cannibals can become hungry) - A rule which causes hungry to be true is required.

```
cross(V) causes hungry if numCrossing(V, strongMissionary) = 0.
```

MCP #17 (cannibals can become hungry) - A rule which causes hungry to be true is required.

```
cross(V) causes hungry if numCrossing(V, strongMissionary) = 0.
```

MCP #10 (a missionary can walk on water) - Rules missing which represent "It cannot be that Jesus is on the boat, but the number of Jesus's on the boat is not 1, or that Jesus is not on the boat and the number of Jesus's is not 0."

```
nonexecutable cross(Vessel) if jesusOnBoat(Vessel) & loading(Vessel, jesus) \= 1.
nonexecutable cross(Vessel) if ~jesusOnBoat(Vessel) & loading(Vessel, jesus) \=0.
```

MCP #10 (a missionary can walk on water) - Updating Jesus's location and count on each side is missing.

```
cross(Vessel) causes jesusLocation = Loc if jesusOnBoat(Vessel) & jesusLocation = Loc1 & Loc\=Loc1.
walk increments numOnBank(jesus, Loc) by 1 if jesusLocation = Loc1 & Loc\=Loc1.
walk decrements numOnBank(jesus, Loc) by 1 if jesusLocation = Loc.
```

**Harmful rules which represent something not specified in the problem.** MCP # 6 (only one missionary and one cannibal can row) - The following rule disallows both the missionary rower and the cannibal rower to be on the boat at the same time. This condition makes it impossible to cross all missionaries and cannibals. Thus it should simply be removed.

```
nonexecutable cross(boat) if rower(boat, missionaries) & rower(boat, cannibals).
```

MCP # 6 (only one missionary and one cannibal can row) - There is no reason to disallow both the missionary and cannibal rower to be on board. The following should be removed.

```
% 13. The crossing action is not executable if both missionaries and cannibals are rowing.
nonexecutable cross(boat) if rower(boat, missionaries) & rower(boat, cannibals).
```

Sudoku (var1) - the following default rule is incorrect. It allows the initial values to be changed in a later step, and changing of the board from unsatisfiable to satisfiable.

```
% Default value of val is none
default val(R, C) = none.
```

MCP #5 (an oar on each bank) - The following constraints are incorrect because the variables N1, N2, and N3 are quantified over all integer values. It should instead be represented with an implication (always (...) → (...)). What is trying to be represented (the natural language writing) is not necessary. Therefore they can be simply removed.

```
always N1 + N2 + N3 = 3 & N1 = numMissionariesAt(bank1) & N2 = numMissionariesAt(bank2) & N3 = numMissionariesOn(boat).
% 11. Total number of cannibals is always 3.
always N1 + N2 + N3 = 3 & N1 = numCannibalsAt(bank1) & N2 = numCannibalsAt(bank2) & N3 = numCannibalsOn(boat).
% 12. Total number of oars is always 2.
always N1 + N2 + N3 = 2 & N1 = numOarsAt(bank1) & N2 = numOarsAt(bank2) & N3 = numOarsOn(boat).
```

MCP #10 (a missionary can walk on water) - The following makes it so the boat's location and Jesus's location cannot change. They should only be set in the query at time 0. These should be removed.

```
% Initial conditions
boatLocation = bank1.
jesusLocation = bank1.
```

**Harmful rules attempting to represent an aspect of the problem.** MCP # 17 (cannibals can become hungry) - The following constraint about cannibals outnumbering missionaries should be changed to include the condition `hungry`. This is because the problem description states that if the cannibals are not hungry, then the missionaries will not be eaten.

```
impossible numAt(missionaries , Loc) = N1 &
numAt(strongMissionary , Loc) = N2 &
numAt(cannibals , Loc) = N3 &
N1 + N2 > 0 &
N3 > N1 + N2 & hungry.
```

MCP #2 (missionaries and cannibals can exchange hats) - Incrementing/decrementing does not work properly due to  $\mathcal{BC}+$  semantics.

```
% For missionaries:
cross(V) decrements numMissionariesAt(bank1) by 1 if onboard(V, Ms) & loc_member(Ms) = bank1 & loc_vessel(V) = bank1.
cross(V) increments numMissionariesAt(bank2) by 1 if onboard(V, Ms) & loc_member(Ms) = bank1 & loc_vessel(V) = bank1.

cross(V) decrements numMissionariesAt(bank2) by 1 if onboard(V, Ms) & loc_member(Ms) = bank2 & loc_vessel(V) = bank2.
cross(V) increments numMissionariesAt(bank1) by 1 if onboard(V, Ms) & loc_member(Ms) = bank2 & loc_vessel(V) = bank2.

% For cannibals:
cross(V) decrements numCannibalsAt(bank1) by 1 if onboard(V, Cs) & loc_member(Cs) = bank1 & loc_vessel(V) = bank1.
cross(V) increments numCannibalsAt(bank2) by 1 if onboard(V, Cs) & loc_member(Cs) = bank1 & loc_vessel(V) = bank1.

cross(V) decrements numCannibalsAt(bank2) by 1 if onboard(V, Cs) & loc_member(Cs) = bank2 & loc_vessel(V) = bank2.
cross(V) increments numCannibalsAt(bank1) by 1 if onboard(V, Cs) & loc_member(Cs) = bank2 & loc_vessel(V) = bank2.
```

These should be corrected to:

```
cross(V) decrements numMissionariesAt(L) by N if numMissionariesOnBoard(V) = N & loc_vessel(V) = L.
cross(V) increments numMissionariesAt(L) by N if numMissionariesOnBoard(V) = N & loc_vessel(V) = L1 & L \= L1.

cross(V) decrements numCannibalsAt(L) by N if numCannibalsOnBoard(V) = N & loc_vessel(V) = L.
cross(V) increments numCannibalsAt(L) by N if numCannibalsOnBoard(V) = N & loc_vessel(V) = L1 & L \= L1.
```

MCP #9 (big cannibal and small missionary) - The disjunction should be encapsulated in parentheses.

```
nonexecutable cross2(P1, P2) if loc(P1) \= loc(boat) | loc(P2) \= loc(boat). % The disjunction should be encapsulated in parentheses
```

MCP #9 (big cannibal and small missionary) - The following laws need to assure that the location where the missionaries/- cannibals are incremented is not where the boat is.

```
cross1(P) increments num_missionaries(NewLoc) by 1 if P = M & NewLoc \= Loc.
cross1(P) increments num_cannibals(NewLoc) by 1 if P = C & NewLoc \= Loc.
```

These should be corrected to:

```
cross1(P) increments num_missionaries(NewLoc) by 1 if P = M & NewLoc \= Loc & loc(boat) = Loc.
cross1(P) increments num_cannibals(NewLoc) by 1 if P = C & NewLoc \= Loc & loc(boat) = Loc.
```

MCP #9 (big cannibal and small missionary) - The term “else” is not recognized in CPLUS2ASP.

```
cross2(P1, P2) decrements num_missionaries(Loc) by L1 if L1 = N1 + N2 & N1 = 1 if P1 = M else N1 = 0 & N2 = 1 if P2 = M else N2 = 0 & loc(P1) =
Loc & loc(P2) = Loc.
cross2(P1, P2) decrements num_cannibals(Loc) by L2 if L2 = N1 + N2 & N1 = 1 if P1 = C else N1 = 0 & N2 = 1 if P2 = C else N2 = 0 & loc(P1) = Loc &
loc(P2) = Loc.
cross2(P1, P2) increments num_missionaries(NewLoc) by L1 if L1 = N1 + N2 & N1 = 1 if P1 = M else N1 = 0 & N2 = 1 if P2 = M else N2 = 0 & NewLoc \=
Loc & loc(P1) = Loc & loc(P2) = Loc.
cross2(P1, P2) increments num_cannibals(NewLoc) by L2 if L2 = N1 + N2 & N1 = 1 if P1 = C else N1 = 0 & N2 = 1 if P2 = C else N2 = 0 & NewLoc \=
Loc & loc(P1) = Loc & loc(P2) = Loc.
```

These should be corrected to:

```
cross2(M1, M2) decrements num_missionaries(Loc) by 2 if loc(boat) = Loc.
cross2(C1, C2) decrements num_cannibals(Loc) by 2 if loc(boat) = Loc.
cross2(M1, C1) decrements num_missionaries(Loc) by 1 if loc(boat) = Loc.
cross2(C1, M1) decrements num_missionaries(Loc) by 1 if loc(boat) = Loc.
```

```

cross2(MI, C1) decrements num.cannibals(Loc) by 1 if loc(boat) = Loc.
cross2(C1, MI) decrements num.cannibals(Loc) by 1 if loc(boat) = Loc.

cross2(MI, M2) increments num.missionaries(Loc) by 2 if loc(boat) = Loc1 & Loc\=Loc1.
cross2(C1, C2) increments num.cannibals(Loc) by 2 if loc(boat) = Loc1 & Loc\=Loc1.
cross2(MI, C1) increments num.missionaries(Loc) by 1 if loc(boat) = Loc1 & Loc\=Loc1.
cross2(C1, MI) increments num.missionaries(Loc) by 1 if loc(boat) = Loc1 & Loc\=Loc1.
cross2(MI, C1) increments num.cannibals(Loc) by 1 if loc(boat) = Loc1 & Loc\=Loc1.
cross2(C1, MI) increments num.cannibals(Loc) by 1 if loc(boat) = Loc1 & Loc\=Loc1.

```

MCP #10 (a missionary can walk on water) - This rules incorrectly disallows crossing if there are more cannibals than missionaries on board, but it should only disallow it if there are more *and* there is at least one missionary on board.

```

% 9b. Crossing the boat is not executable if cannibals loaded onto the boat exceed missionaries loaded onto the boat if jesusOnBoat(Vessel) = false.

nonexecutable cross(Vessel) if loading(Vessel, cannibals) = N_c & loading(Vessel, missionaries) = N_m & jesusOnBoat(Vessel) = false & N_c > N_m.

% N_m > 0 should be added to the if clause

```

MCP #10 (a missionary can walk on water) - Similarly, there being at least one missionary needs to be adding to the body of these rules to make them correct.

```

% 10b. Crossing the boat is not executable if cannibals remaining on the starting bank after loading would outnumber missionaries remaining on the starting bank when Jesus is not at that bank.

nonexecutable cross(Vessel) if boatLocation = Loc & jesusLocation \= Loc &
numOnBank(cannibals, Loc) = N_c & loading(Vessel, cannibals) = L_c & Rem_c = N_c - L_c &
numOnBank(missionaries, Loc) = N_m & loading(Vessel, missionaries) = L_m & Rem_m = N_m - L_m &
Rem_c > Rem_m.

% Rem_m > 0, needs to be added to the if clause

```

MCP #10 (a missionary can walk on water) - Similarly, there being at least one missionary needs to be adding to the body of this rule to make it correct.

```

It should be possible for the number of cannibals to be larger than the number of missionaries if there are no missionaries on the bank generated.

% 14b. It is impossible for cannibals to outnumber missionaries on any bank when Jesus is not at that bank.

impossible N_c > N_m & numOnBank(cannibals, Loc) = N_c & numOnBank(missionaries, Loc) = N_m & jesusLocation \= Loc.

% N_m > 0, needs to be added to the if clause

```

### D.3 Query

**Syntax issues.** Sudokul - The following query is written with incorrect keywords “initially” and “goal” to represent the initial and goal state of Sudoku.

```

initially valueAt(0,0)=8.
initially valueAt(1,2)=3.
initially valueAt(1,3)=6...
goal valueAt(R,C)\=0.

```

Sudoku (var2) - “maxstep: 81” is not syntactically correct.

```

:- query
0:
val(1,2)=9 & val(1,4)=3 & val(1,9)=1 &
val(2,5)=8 & val(2,8)=4 & val(2,9)=6 &
val(3,7)=8 &
val(4,1)=4 & val(4,3)=5 & val(4,5)=6 & val(4,8)=3 &
val(5,3)=3 & val(5,4)=2 & val(5,5)=7 & val(5,6)=5 & val(5,7)=6 &
val(6,2)=6 & val(6,5)=1 & val(6,7)=9 & val(6,9)=4 &
val(7,3)=1 &
val(8,1)=5 & val(8,2)=8 & val(8,5)=2 &
val(9,1)=2 & val(9,6)=7 & val(9,8)=6;
maxstep: 81. % should be "maxstep :: 81.", alternatively it can be removed)

```

**Semantic issues.** Tower of Hanoi (7, var1) - The following query goal condition incorrectly states that that every disk should be on peg c. Instead, they should be stacked in order, with only disk d1 on c.

```

:- query
0: loc(d1)=c & loc(d2)=d1 & loc(d3)=d2 & loc(d4)=a & loc(d5)=b & loc(d6)=d4 & loc(d7)=d5 &
 weight(d1)=7 & weight(d2)=6 & weight(d3)=5 & weight(d4)=4 & weight(d5)=3 & weight(d6)=2 & weight(d7)=1;
maxstep: loc(d1)=c & loc(d2)=c & loc(d3)=c & loc(d4)=c & loc(d5)=c & loc(d6)=c & loc(d7)=c.

```

% The maxstep should be as follows:

```

maxstep: loc(d1)=c & loc(d2)=d1 & loc(d3)=d2 & loc(d4)=d3 &
loc(d5)=d4 & loc(d6)=d5 & loc(d7)=d6.

```

MCP #3 (there are 4 missionaries and 4 cannibals) - There are two issues: (1) The query includes a maxstep of 11, which it should not, (2) the query uses 3 for the number of missionaries and cannibals, this should be 4.

```

:- query
0: numOn(bank1, missionaries) = 3 & numOn(bank1, cannibals) = 3 & loc(boat) = bank1 & numOn(bank2, missionaries) = 0 & numOn(bank2,
 cannibals) = 0;
maxstep :: 11; % (should be removed)
maxstep: numOn(bank2, missionaries) = 3 & numOn(bank2, cannibals) = 3.

```

## E Missionaries and Cannibals Elaborations

In McCarthy (1998), there are 20 elaborations discussed in which we include 16 in this work, as well as the original problem:

“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?”

#1 “The boat is a rowboat. (Or the boat is a motorboat). By itself this is a trivial elaboration. Adding it should not affect the reasoning. By default, a tool, i.e. the boat, is usable. Further elaborations might use specific properties of rowboats.”

#2 “The missionaries and cannibals have hats, all different—another trivial elaboration. These hats may be exchanged among the missionaries and cannibals.”

#3 “There are four missionaries and four cannibals. The problem is now unsolvable.”

#4 - “The boat can carry three.” Five missionaries and cannibals can cross, but not six. We use the version with six members of each group, which no solution is possible.

#5 - “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.”

#6 - “Only one missionary and one cannibal can row.”

#7 - “The missionaries can’t row. This makes the problem impossible, since any solution requires two missionaries in the boat at some time”

#8 - “The biggest cannibal cannot fit in the boat with another person.”

#9 - “If the biggest cannibal is isolated with the smallest missionary, the latter will be eaten.”

#10 - “One of the missionaries is Jesus Christ.” This makes it so that one of the missionaries can walk on water, which has a solution with only 7 steps.

#11 - “Three missionaries alone with a cannibal can convert him into a missionary.”

#13 - “There is a bridge. ” This makes makes it so any number can cross, we use the version with 5 missionaries and cannibals.

#14 - “The boat leaks and must be bailed concurrently with rowing.” We use the version from Lifschitz (2000) which assumes that bailing is only needed when there are two people in the boat, and that one person could not row and bail at the same time.

#16 - “There is an island. ” This makes it so any number can cross. We use an instance with 5 missionaries and cannibals.

#17 - “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 has some vagueness, we make it more concrete by specifying that as long as the strongest missionary rows then the cannibals will not be hungry.

#19 - “There are two sets of missionaries and cannibals too far apart along the river to interact. The two problem should be solvable separately without considering interleaving actions at the two sites.” Here we assume that the two sets are two identical versions of the original problem.