Planning in Artificial Intelligence

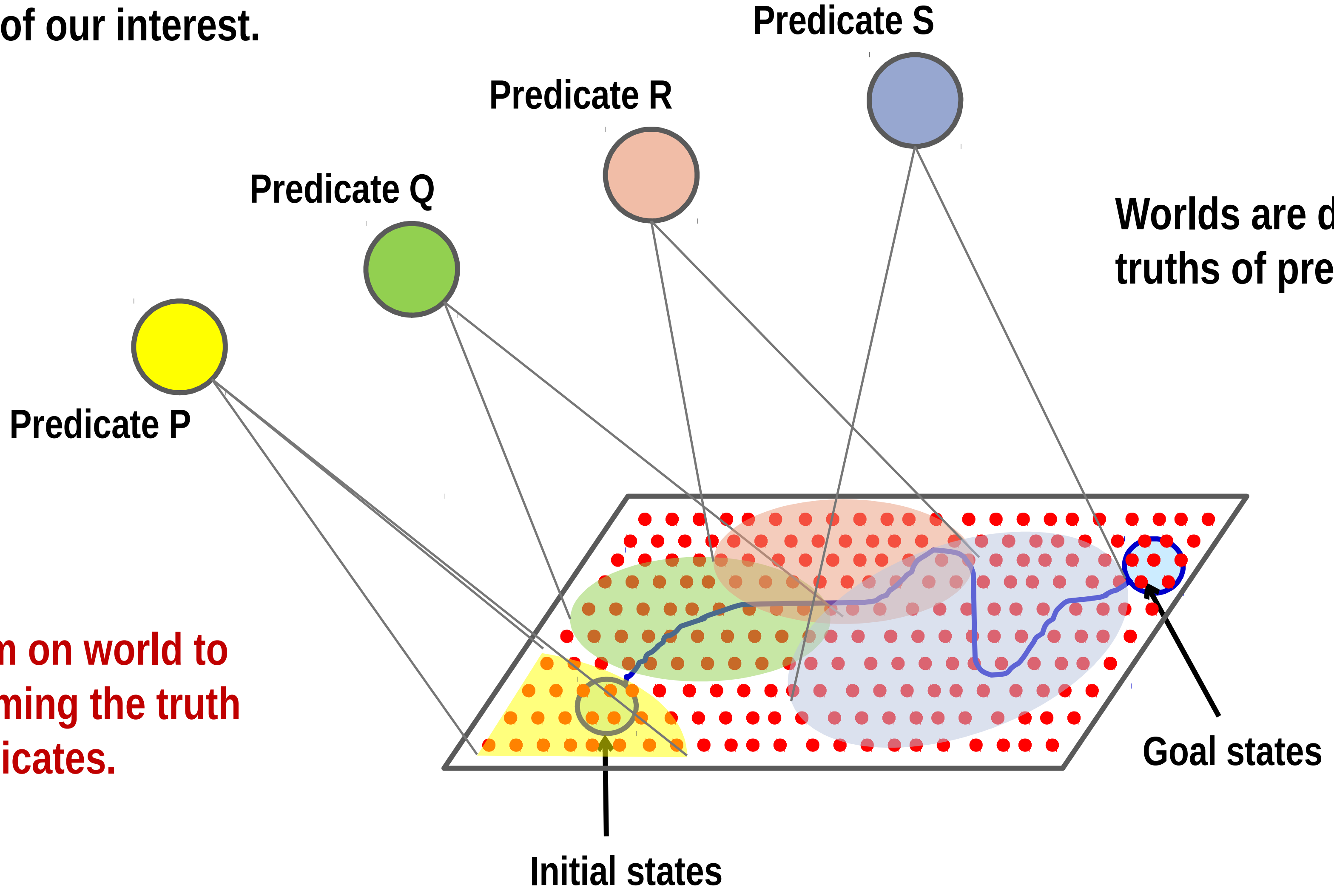
The intelligent way to do things

COURSE: CS60045

Pallab Dasgupta
Professor,
Dept. of Computer Sc & Engg

From State Spaces to Predicate Worlds

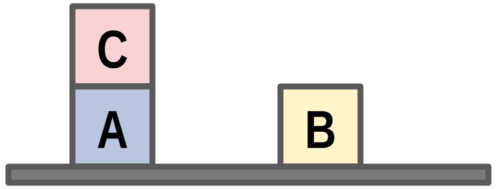
We abstract out the state space in terms of predicates of our interest.



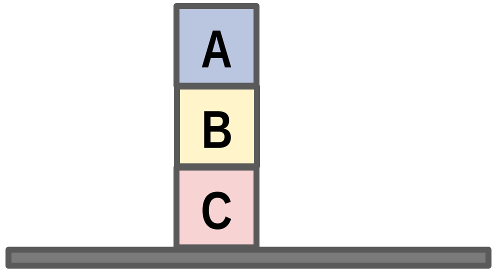
Worlds are defined in terms of truths of predicates.

Actions take us from on world to another by transforming the truth of one or more predicates.

Blocks World



Initial State



Target State

Predicates describing the initial state:
On(C, A), On(A, Table),
On(B, Table),
Clear(C), Clear(B)

Predicates describing the target state:
On(A, B), On(B, C)

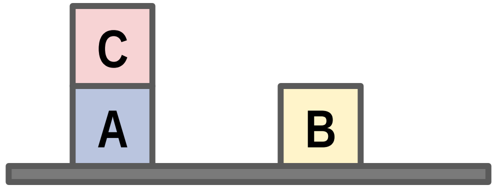
ACTIONS:

Move(X, Y)
Precond: Clear(X), Clear(Y)
Effect: On(X, Y)

Move(X, Table)
Precond: Clear(X)
Effect: On(X, Table)

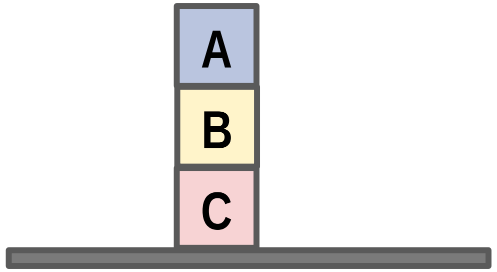
The planning task is to determine the actions for reaching the target state from the initial state.

Choosing Actions



On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

On(A, B), On(B, C)



ACTIONS:

Move(X, Y)
Precond: Clear(X), Clear(Y)
Effect: On(X, Y)

Move(X, Table)
Precond: Clear(X)
Effect: On(X, Table)

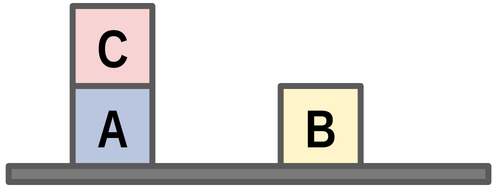
- We can move C to the table
  - This achieves none of the goal predicates
- We can move C to top of B
  - This achieves none of the goal predicates
- We can move B to top of C
  - This achieves On(B, C)

Partial Solutions

ACTIONS:

Move(X, Y)
Precond: Clear(X), Clear(Y)
Effect: On(X, Y)

Move(X, Table)
Precond: Clear(X)
Effect: On(X, Table)

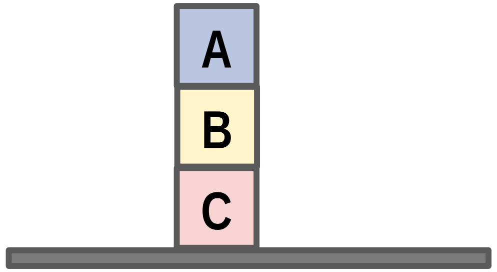


On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

Clear(C), Clear(B)

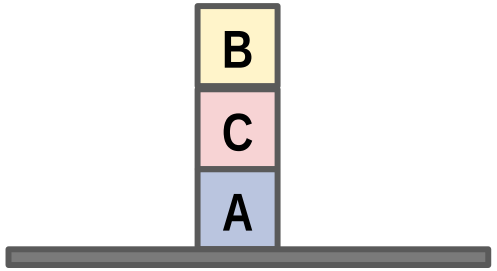
Move(B, C)

On(A, B), On(B, C)



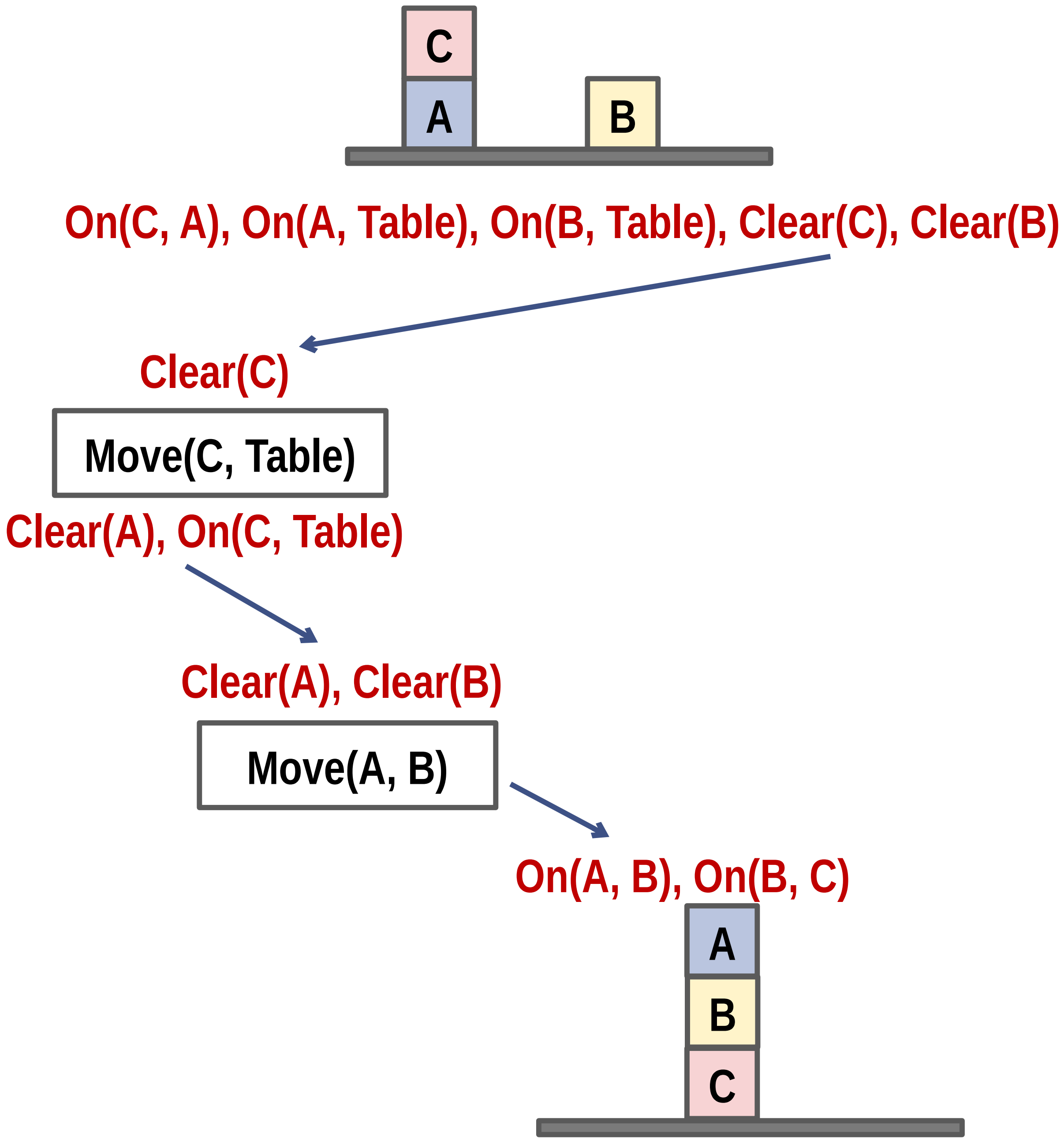
We use Move(B, C) to achieve the sub-goal, On(B, C).

But if we apply this move at the beginning, we get:



Which is not what we want !!

Partial Solutions



On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

Clear(C)

Move(C, Table)

Clear(A), On(C, Table)

Clear(A), Clear(B)

Move(A, B)

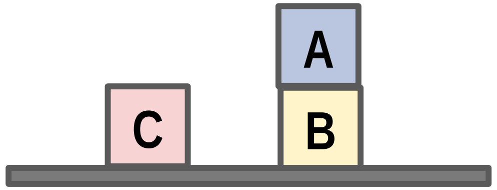
On(A, B), On(B, C)

ACTIONS:

Move(X, Y)
Precond: Clear(X), Clear(Y)
Effect: On(X, Y)

Move(X, Table)
Precond: Clear(X)
Effect: On(X, Table)

The sub-goal On(A, B) is achieved by moving C to the table and then moving A to top to B. But this gives us:



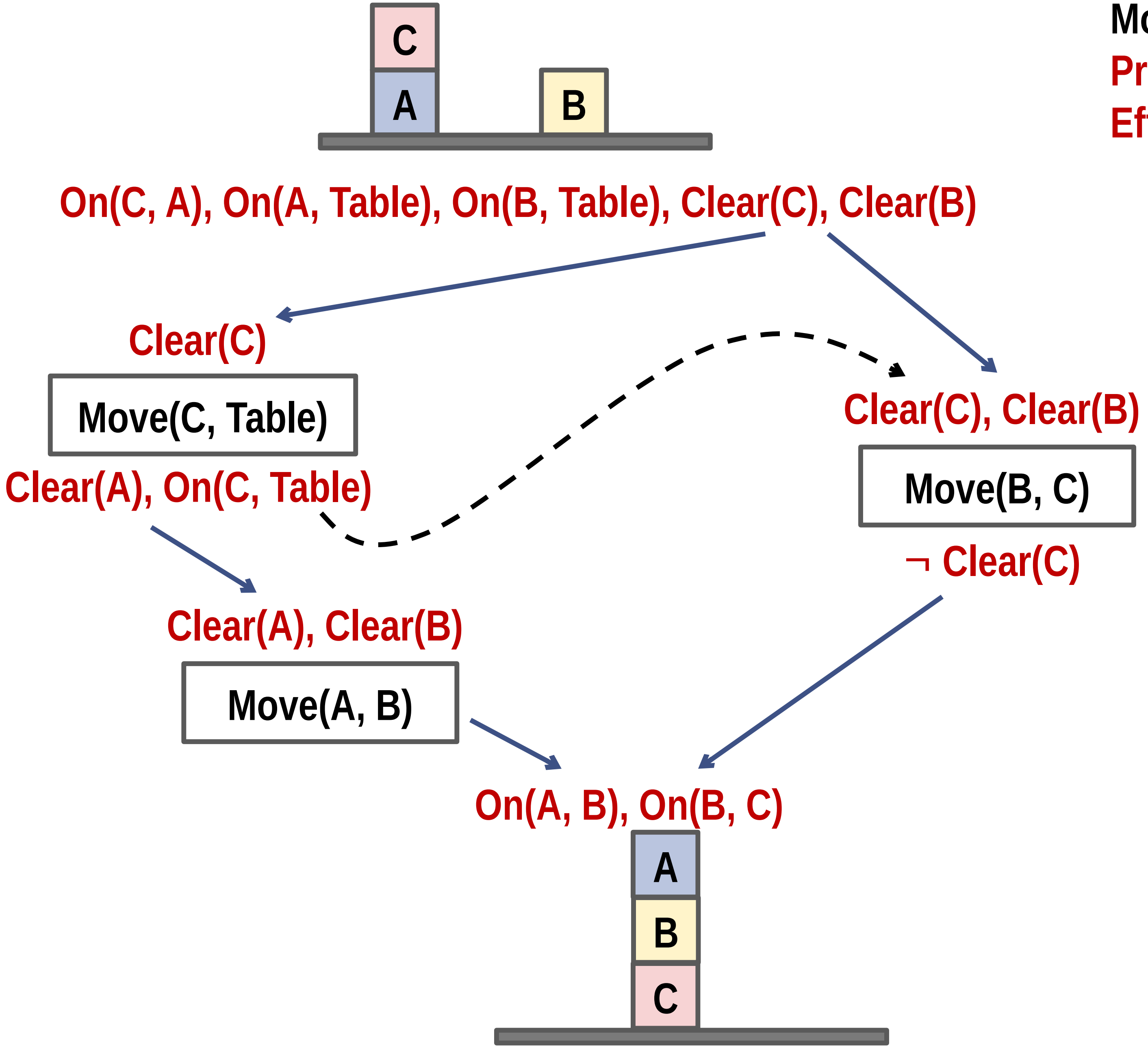
But this too is not what we want !!

Ordering Partial Solutions

ACTIONS:

Move(X, Y)
Precond: Clear(X), Clear(Y)
Effect: On(X, Y)

Move(X, Table)
Precond: Clear(X)
Effect: On(X, Table)



On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

Clear(C)

Move(C, Table)

Clear(A), On(C, Table)

Clear(C), Clear(B)

Move(B, C)

¬ Clear(C)

Clear(A), Clear(B)

Move(A, B)

On(A, B), On(B, C)

Move(B, C) removes the Clear(C) predicate which is essential for Move(C, Table). Hence Move(C, Table) must precede Move(B, C).

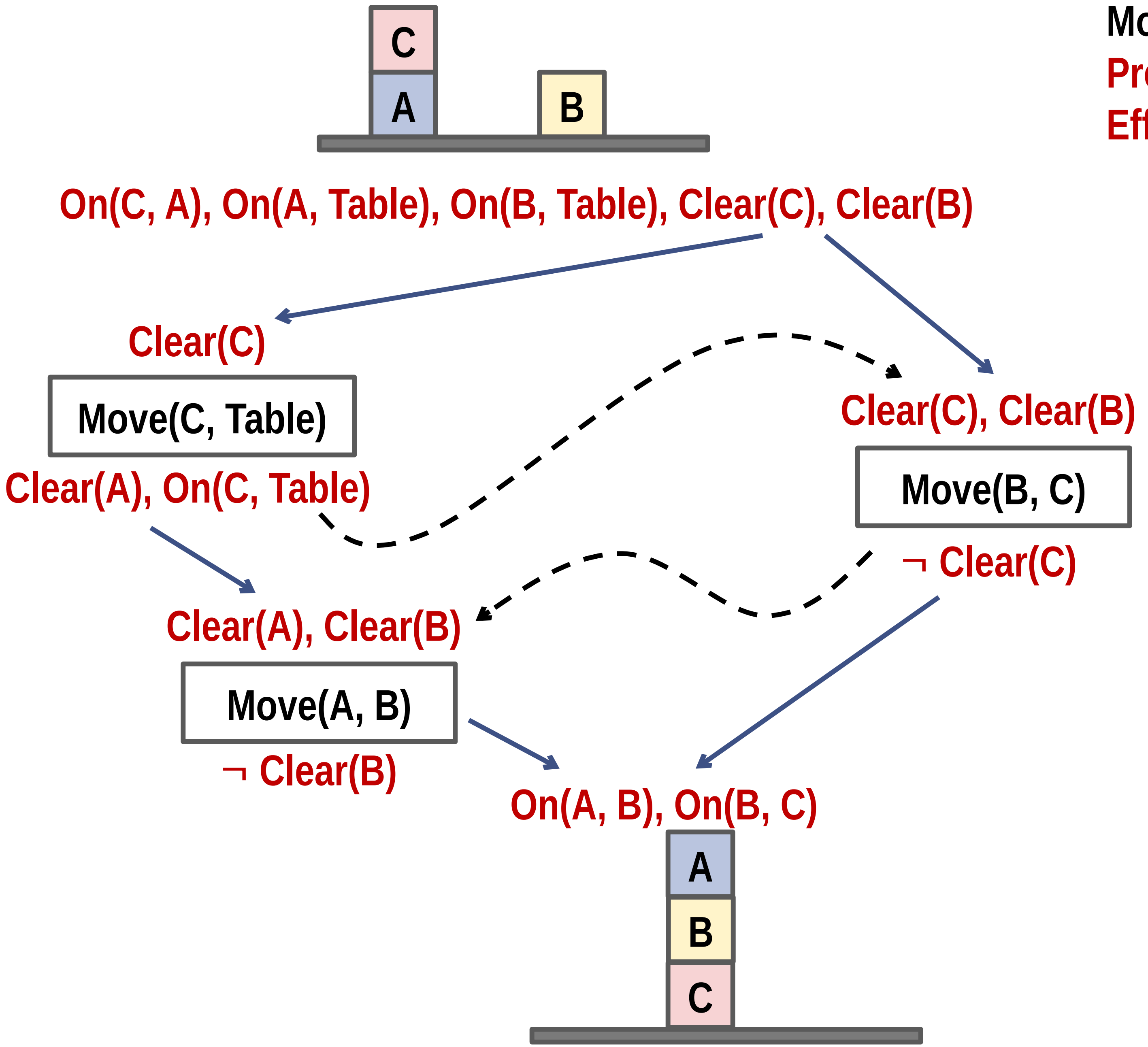
Can Move(B, C) and Move(A, B) be executed in any order?

Ordering Partial Solutions

ACTIONS:

Move(X, Y)
Precond: Clear(X), Clear(Y)
Effect: On(X, Y)

Move(X, Table)
Precond: Clear(X)
Effect: On(X, Table)



On(C, A), On(A, Table), On(B, Table), Clear(C), Clear(B)

Clear(C)

Move(C, Table)

Clear(A), On(C, Table)

Clear(C), Clear(B)

Move(B, C)

¬ Clear(C)

Clear(A), Clear(B)

Move(A, B)

¬ Clear(B)

On(A, B), On(B, C)

Move(A, B) removes the Clear(B) predicate which is essential for Move(B, C). Hence Move(B, C) must precede Move(A, B).

Therefore the only total order is:

1. Move(C, Table)
2. Move(B, C)
3. Move(A, B)

Sometimes Partial Order may stay

ACTIONS

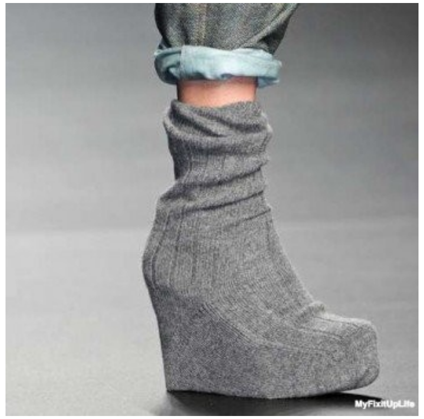
Op(ACTION: RightShoe,
PRECOND::RightSockOn,
EFFECT:: RightShoeOn)

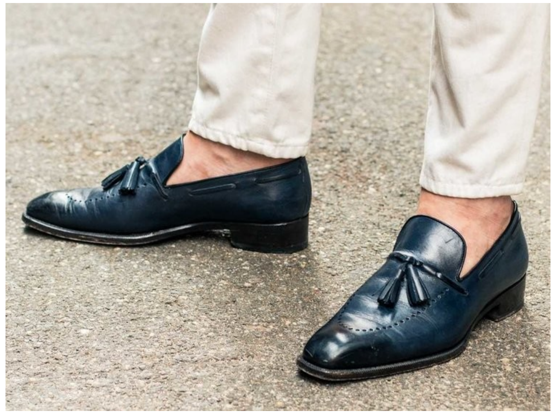
Op(ACTION: RightSock,
EFFECT: RightSockOn)

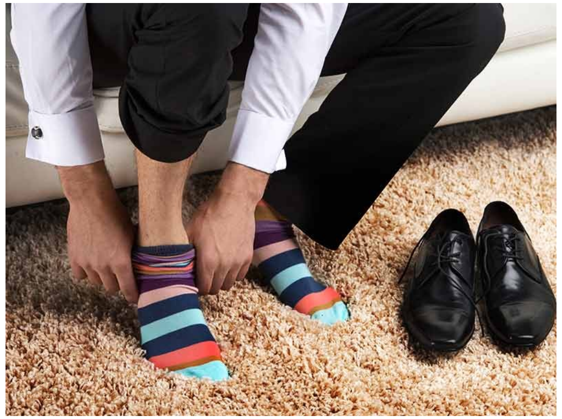
Op(ACTION: LeftShoe,
PRECOND: LeftSockOn,
EFFECT: LeftShoeOn)

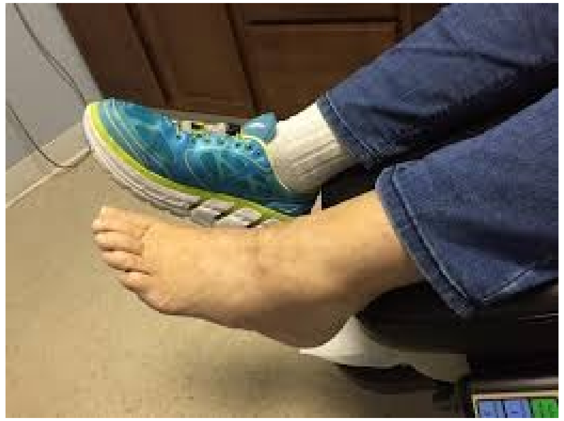
Op(ACTION: LeftSock,
EFFECT: LeftSockOn)

Which of these situations are allowed by these actions?









Sometimes Partial Order may stay

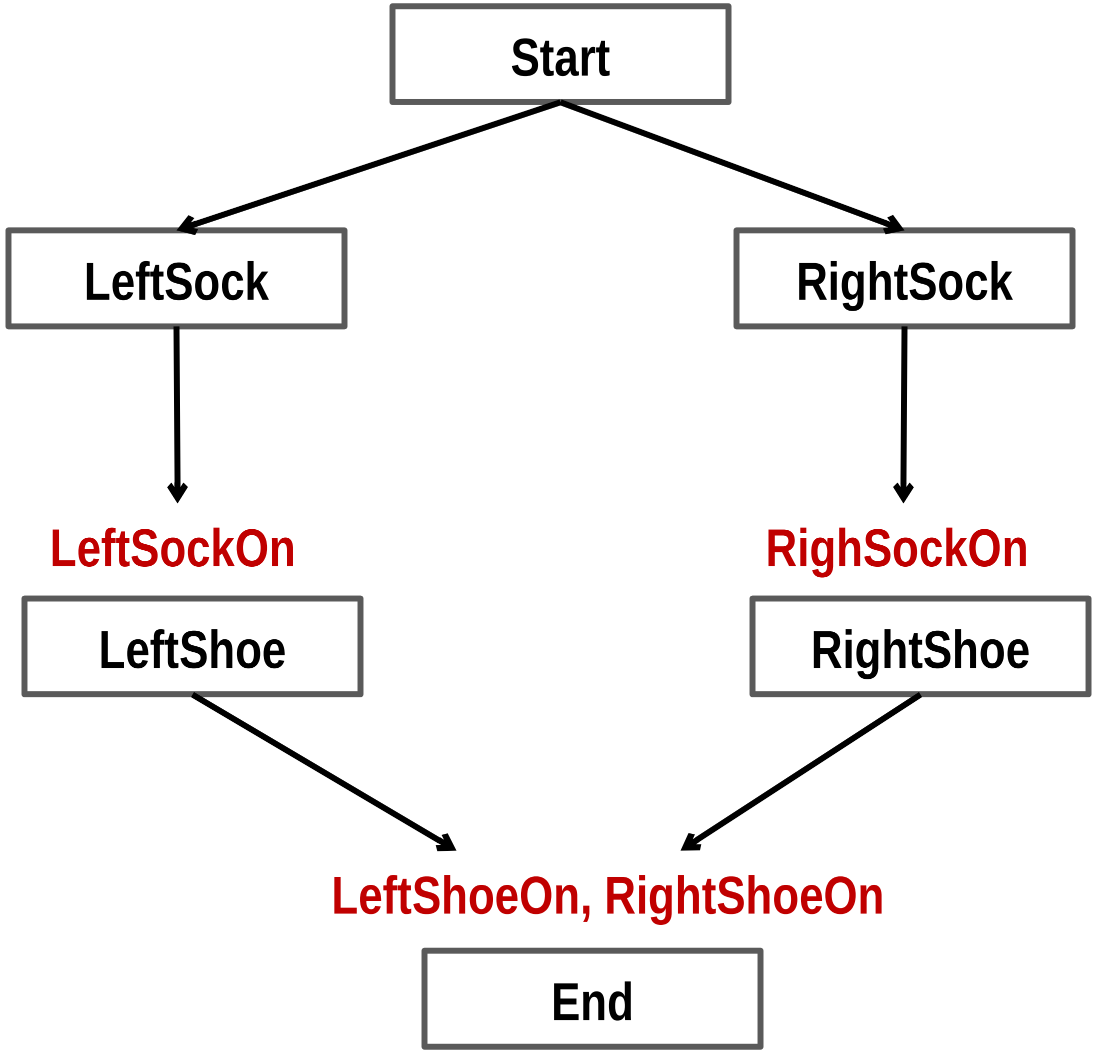
ACTIONS

Op(ACTION: RightShoe,
PRECOND::RightSockOn,
EFFECT:: RightShoeOn)

Op(ACTION: RightSock,
EFFECT: RightSockOn)

Op(ACTION: LeftShoe,
PRECOND: LeftSockOn,
EFFECT: LeftShoeOn)

Op(ACTION: LeftSock,
EFFECT: LeftSockOn)



Example

- Initial plan

```
Plan(
   STEPS: {
         S1: Op(  ACTION: start ),
         S2: Op(  ACTION: finish,
                   PRECOND: RightShoeOn ^ LeftShoeOn )
           },
   ORDERINGS: {S₁ ≺ S₂},
   BINDINGS: { },
   LINKS: { }    )
```

POP Example: Get Tea, Biscuits, Book

Initial state:

Op(ACTION: Start,
EFFECT: At(Home) ^ Sells(BS, Book)
^ Sells(TS, Tea)
^ Sells(TS, Biscuits))

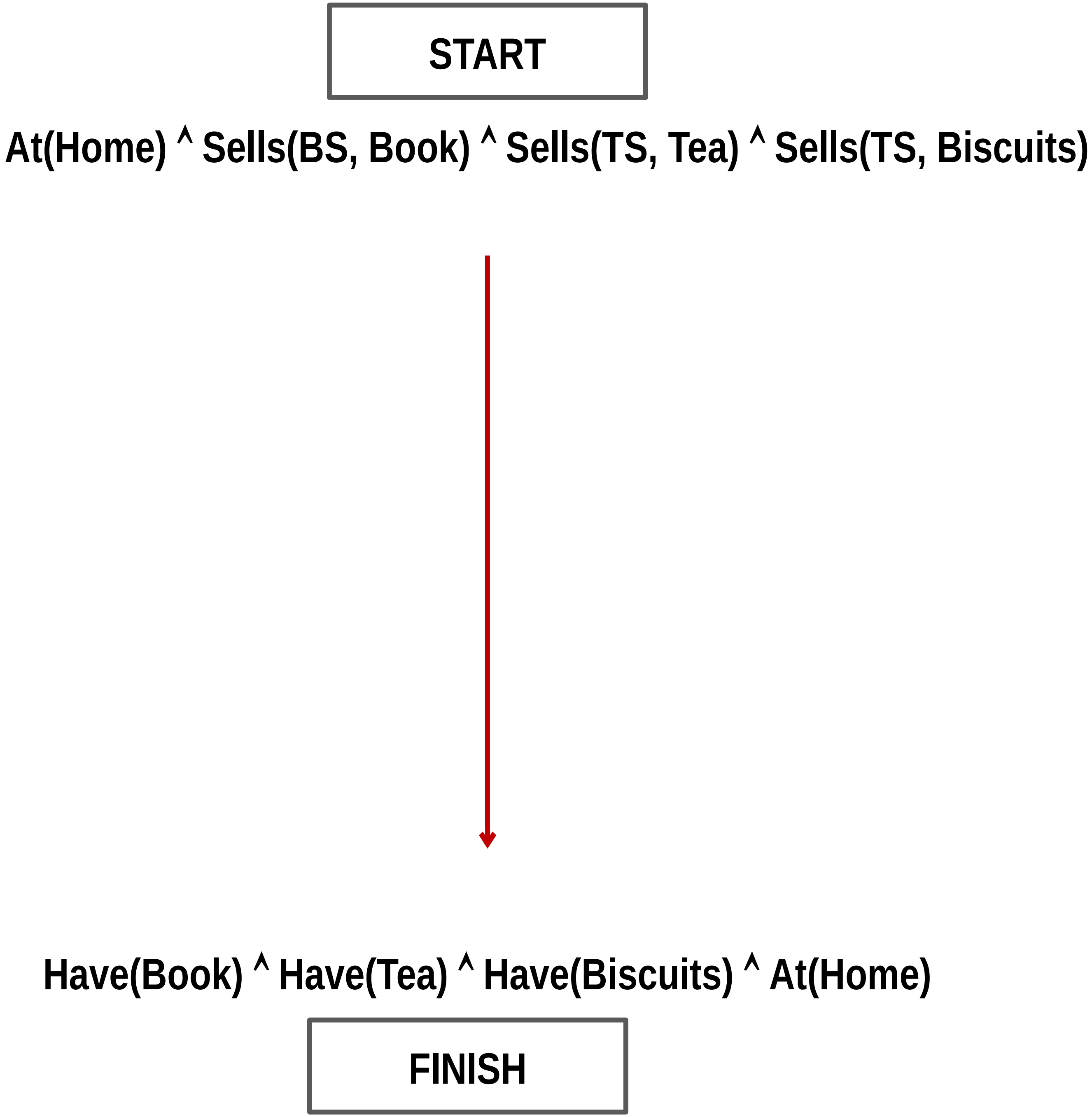
Goal state:

Op(ACTION: Finish,
PRECOND: At(Home) ^ Have(Tea)
^ Have(Biscuits)
^ Have(Book))

Actions:

Op(ACTION: Go(y),
PRECOND: At(x),
EFFECT: At(y) ^ ¬At(x))

Op(ACTION: Buy(x),
PRECOND: At(y) ^ Sells(y, x),
EFFECT: Have(x))

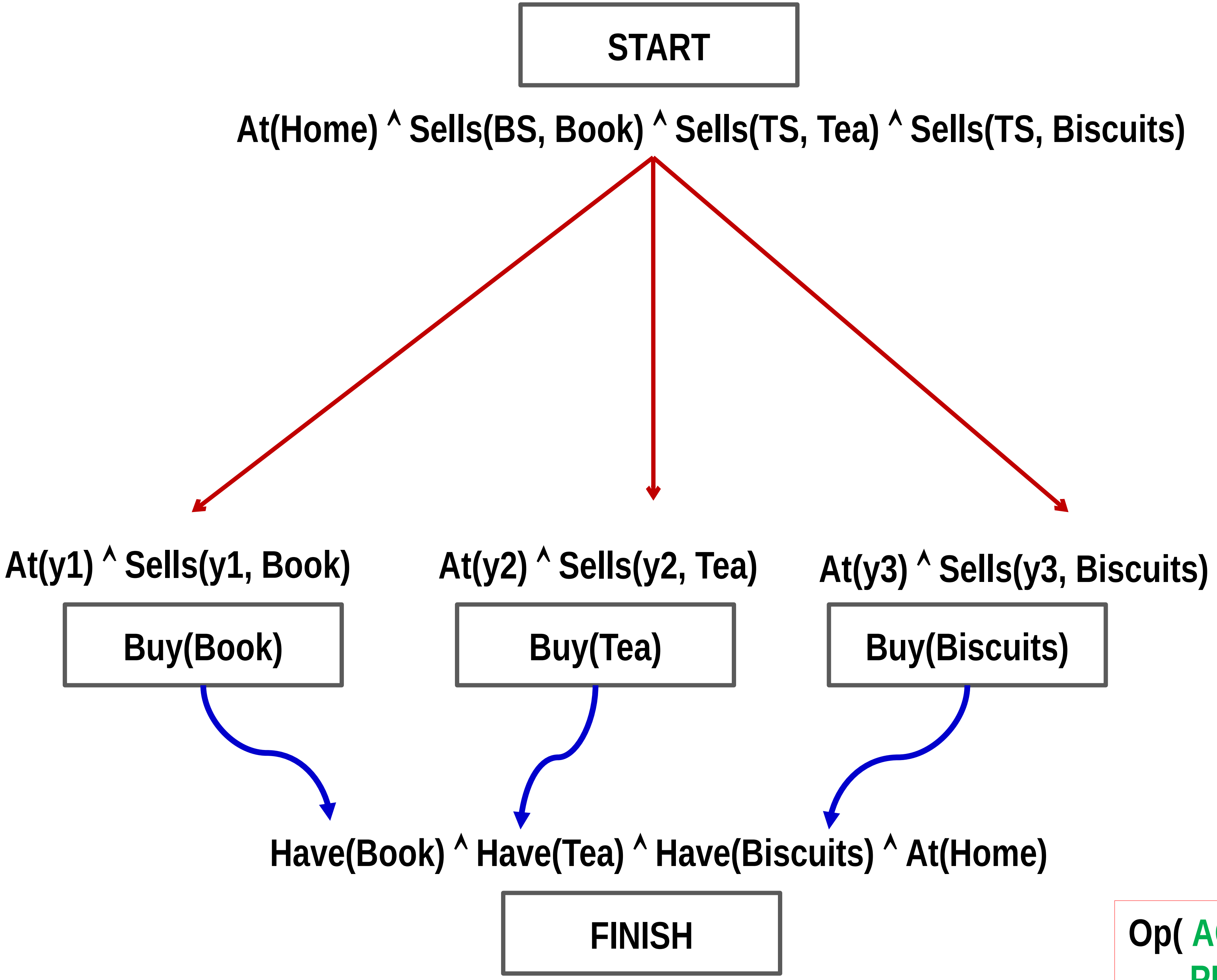


START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(y1) ^ Sells(y1, Book)

Buy(Book)

At(y2) ^ Sells(y2, Tea)

Buy(Tea)

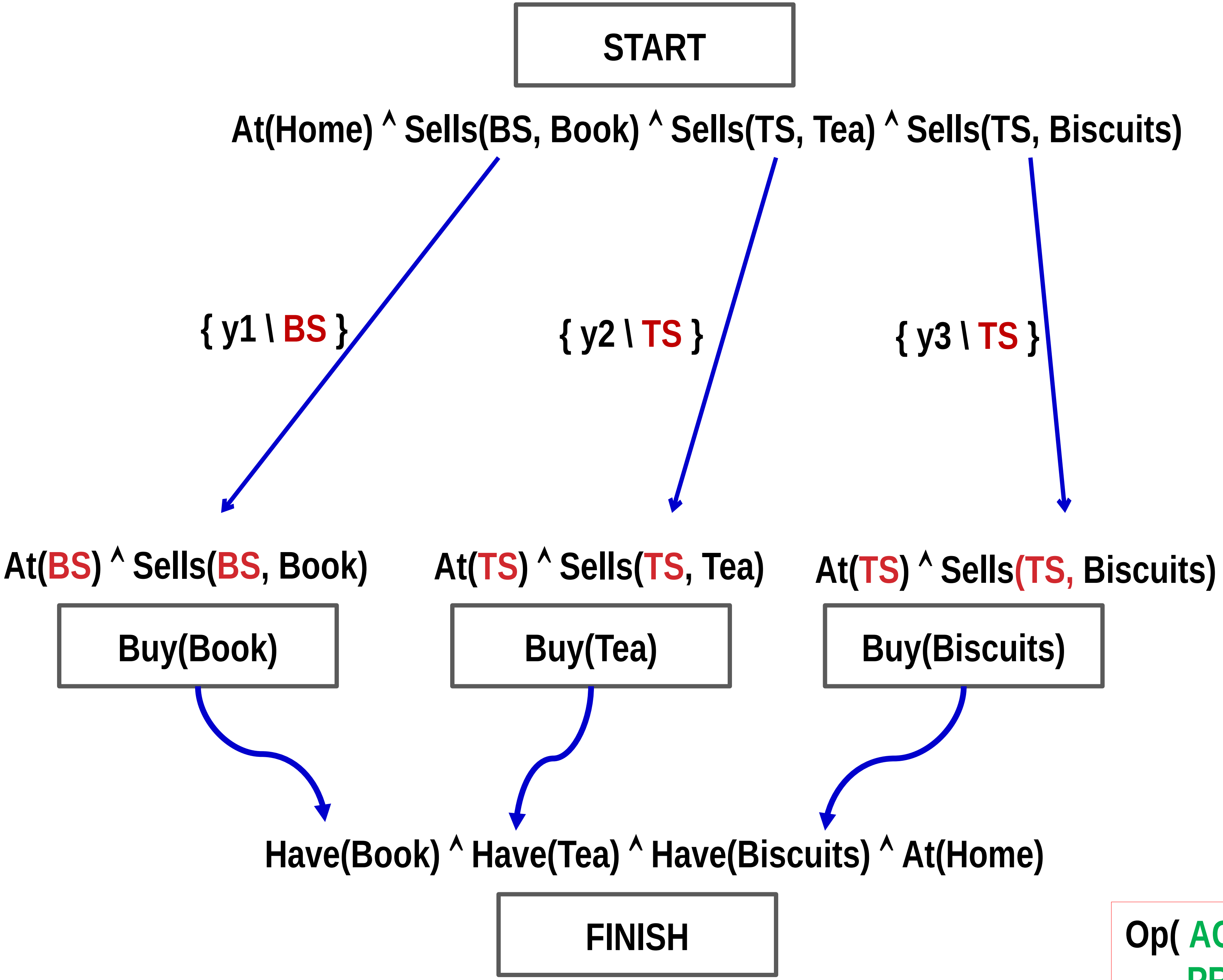
At(y3) ^ Sells(y3, Biscuits)

Buy(Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH

Op(ACTION: Buy(x),
PRECOND: At(y) ^ Sells(y, x),
EFFECT: Have(x))



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

{ y1 \ BS }

{ y2 \ TS }

{ y3 \ TS }

At(BS) ^ Sells(BS, Book)

At(TS) ^ Sells(TS, Tea)

At(TS) ^ Sells(TS, Biscuits)

Buy(Book)

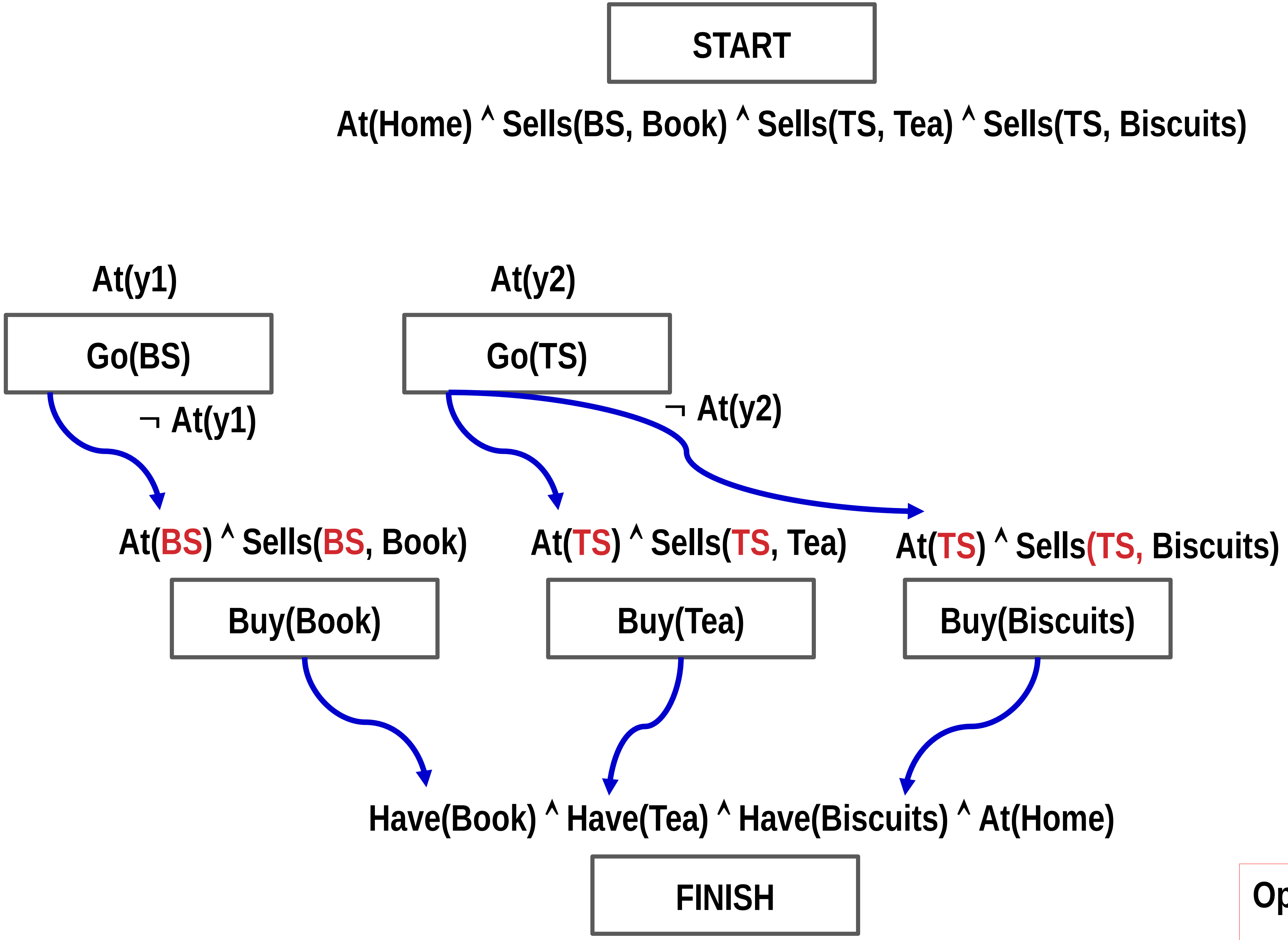
Buy(Tea)

Buy(Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH

Op(ACTION: Buy(x),
PRECOND: At(y) ^ Sells(y, x),
EFFECT: Have(x))



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(y1)

Go(BS)

¬ At(y1)

At(y2)

Go(TS)

¬ At(y2)

At(BS) ^ Sells(BS, Book)

Buy(Book)

At(TS) ^ Sells(TS, Tea)

Buy(Tea)

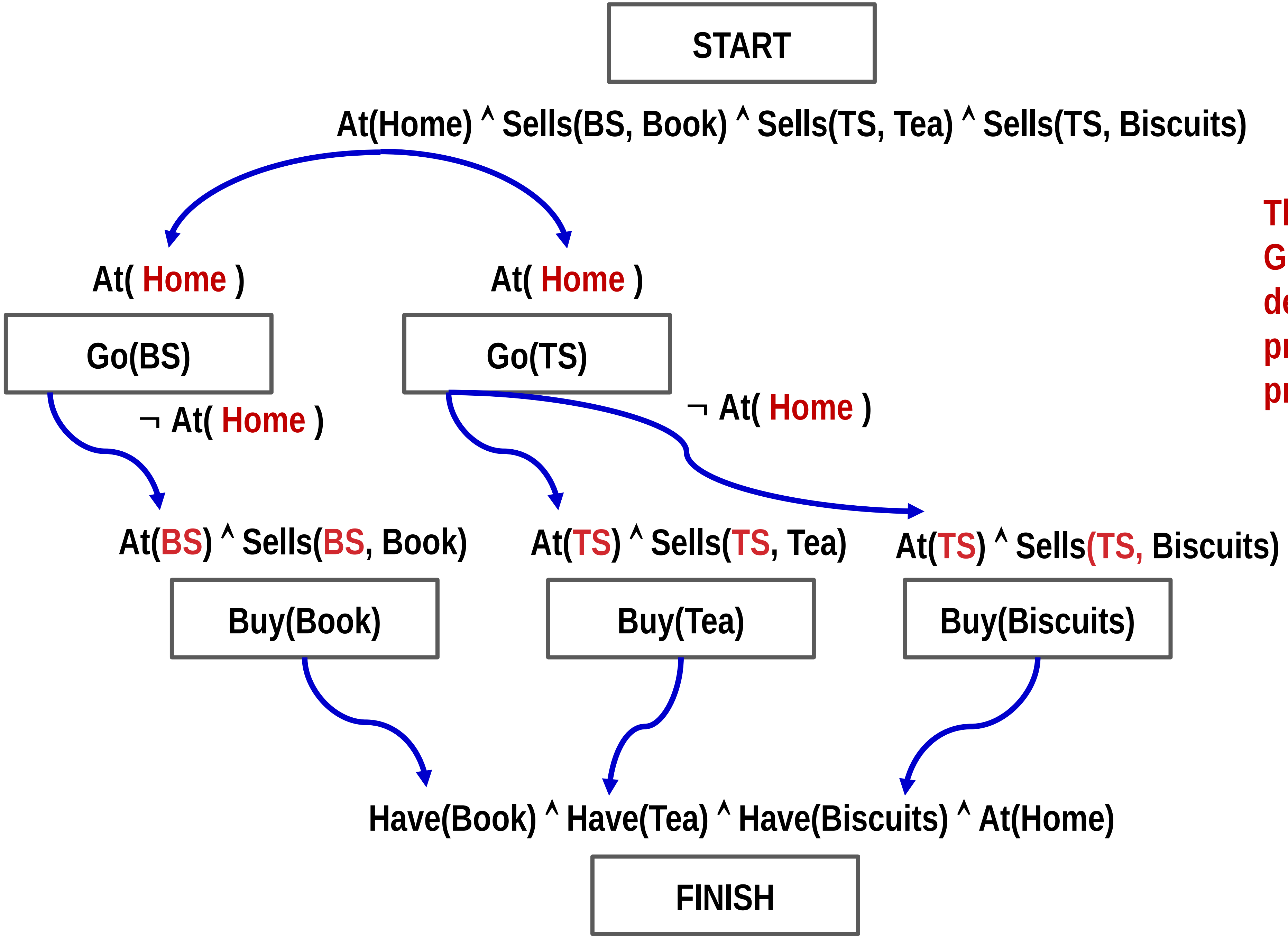
At(TS) ^ Sells(TS, Biscuits)

Buy(Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH

Op(ACTION: Go(y),
PRECOND: At(x),
EFFECT: At(y) ^ ¬At(x))



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(Home)

At(Home)

Go(BS)

Go(TS)

¬ At(Home)

¬ At(Home)

At(BS) ^ Sells(BS, Book)

At(TS) ^ Sells(TS, Tea)

At(TS) ^ Sells(TS, Biscuits)

Buy(Book)

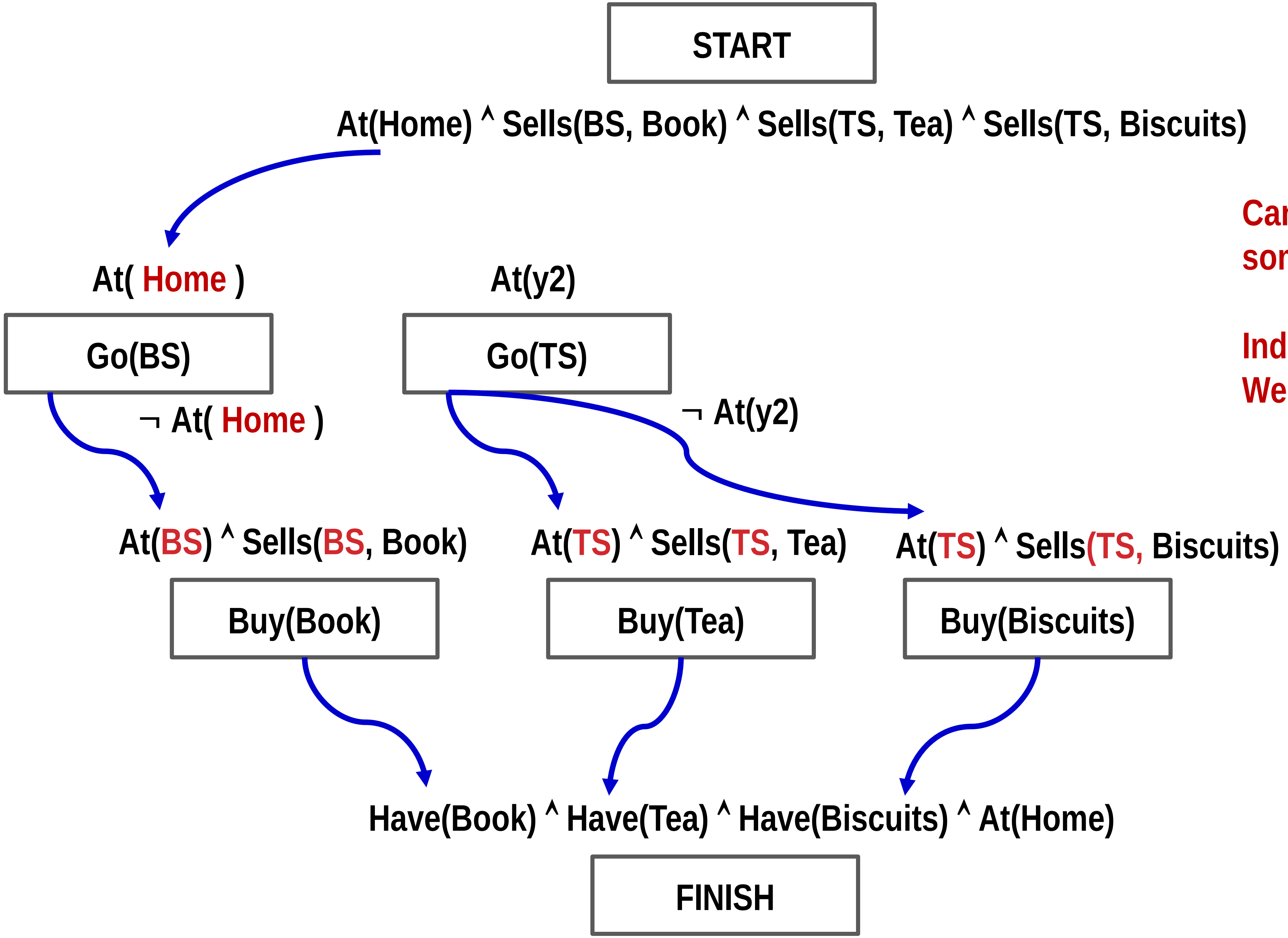
Buy(Tea)

Buy(Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH

The problem here is that Go(BS) and Go(TS) destroy each other's precondition. Neither can precede the other.



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(Home)

Go(BS)

¬ At(Home)

At(y2)

Go(TS)

¬ At(y2)

At(BS) ^ Sells(BS, Book)

Buy(Book)

At(TS) ^ Sells(TS, Tea)

Buy(Tea)

At(TS) ^ Sells(TS, Biscuits)

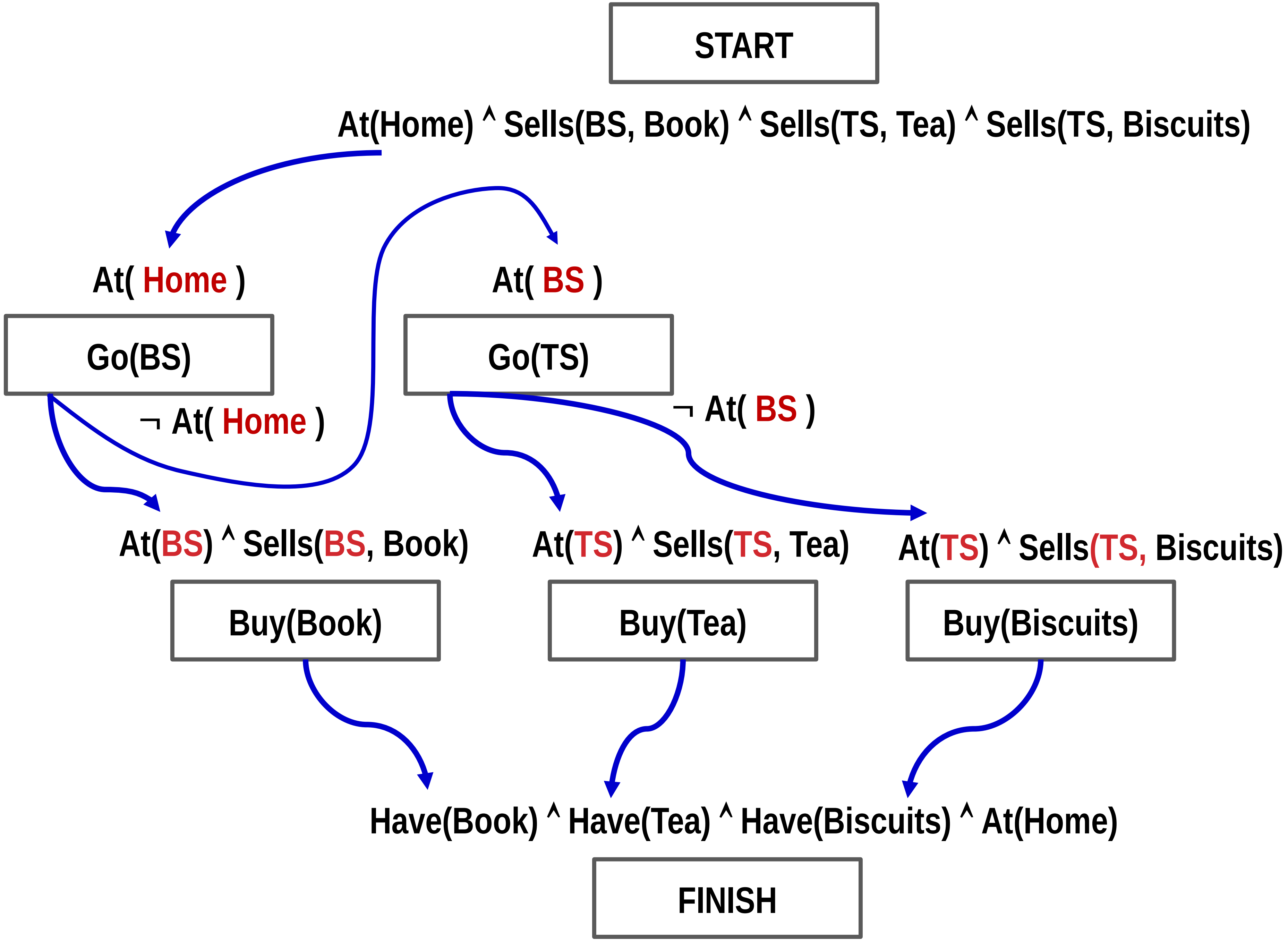
Buy(Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH

Can y2 be instantiated with something else?

Indeed !!
We can try BS for example.



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(Home)

Go(BS)

¬ At(Home)

At(BS)

Go(TS)

¬ At(BS)

At(BS) ^ Sells(BS, Book)

Buy(Book)

At(TS) ^ Sells(TS, Tea)

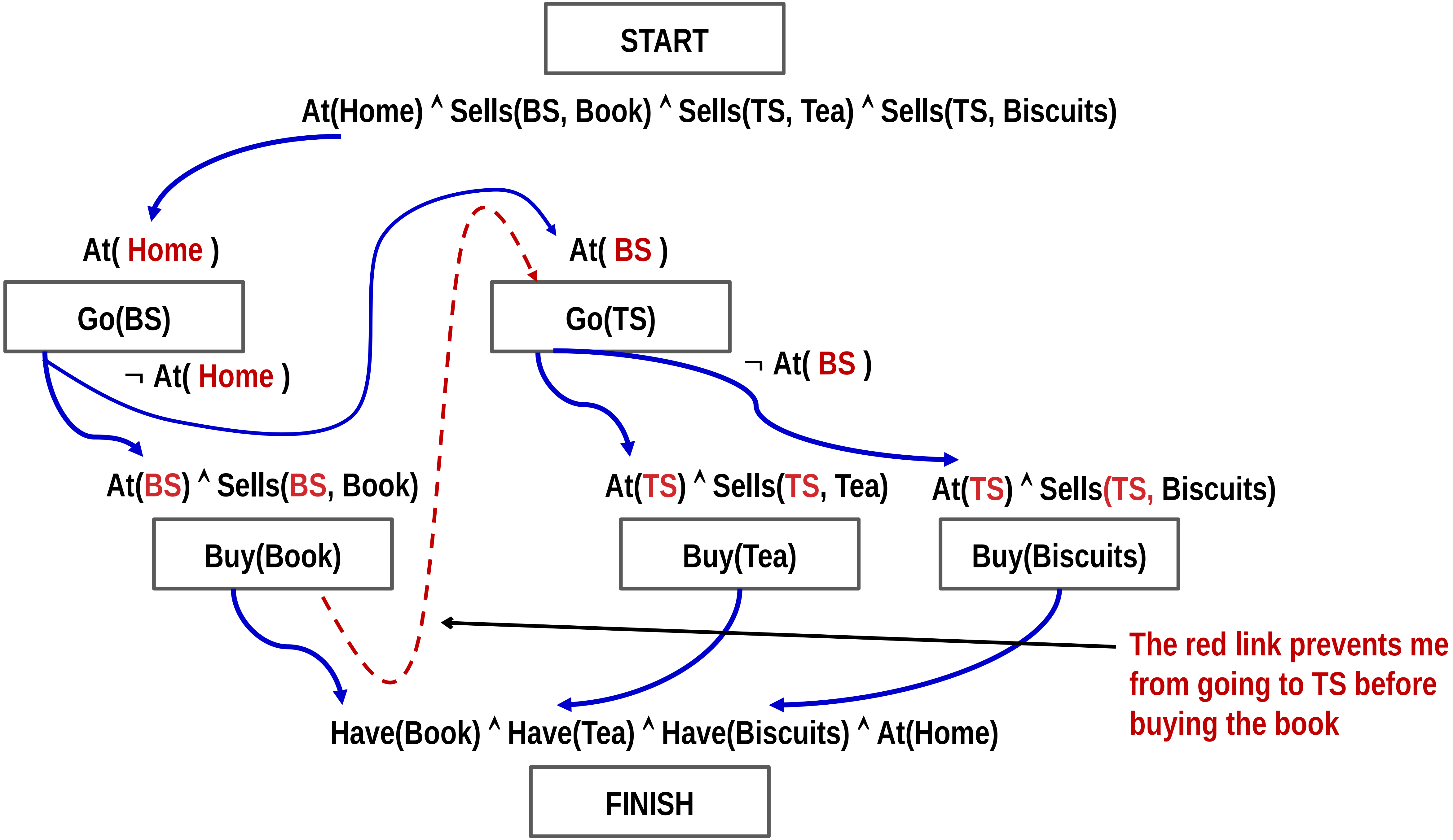
Buy(Tea)

At(TS) ^ Sells(TS, Biscuits)

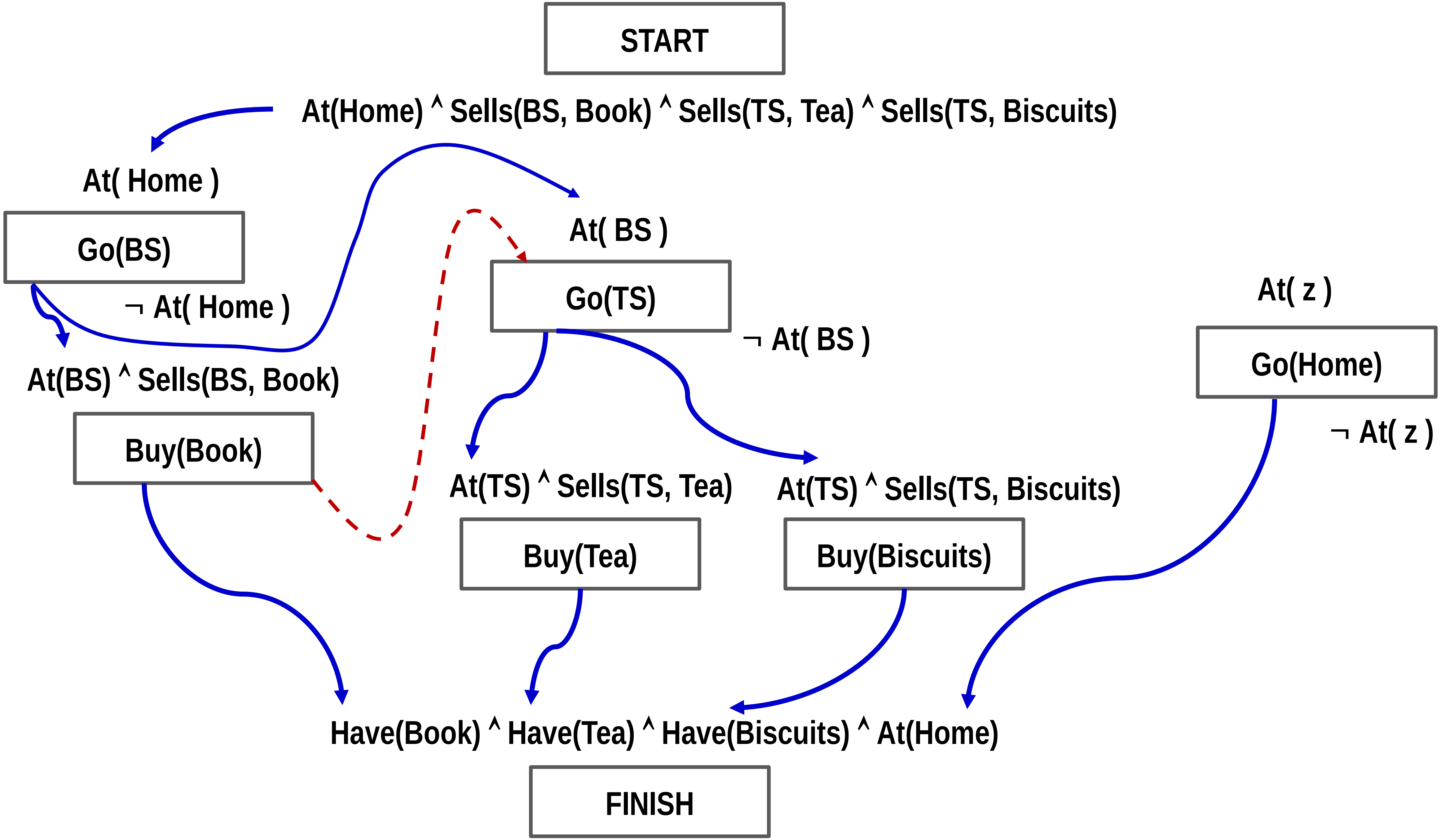
Buy(Biscuits)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH


START
At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)
At(Home)
At(BS)
Go(BS)
Go(TS)
¬ At(Home)
¬ At(BS)
At(BS) ^ Sells(BS, Book)
At(TS) ^ Sells(TS, Tea)
At(TS) ^ Sells(TS, Biscuits)
Buy(Book)
Buy(Tea)
Buy(Biscuits)
Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)
FINISH

The red link prevents me from going to TS before buying the book



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(Home)

Go(BS)

¬ At(Home)

At(BS) ^ Sells(BS, Book)

Buy(Book)

At(BS)

Go(TS)

¬ At(BS)

At(TS) ^ Sells(TS, Tea)

Buy(Tea)

At(TS) ^ Sells(TS, Biscuits)

Buy(Biscuits)

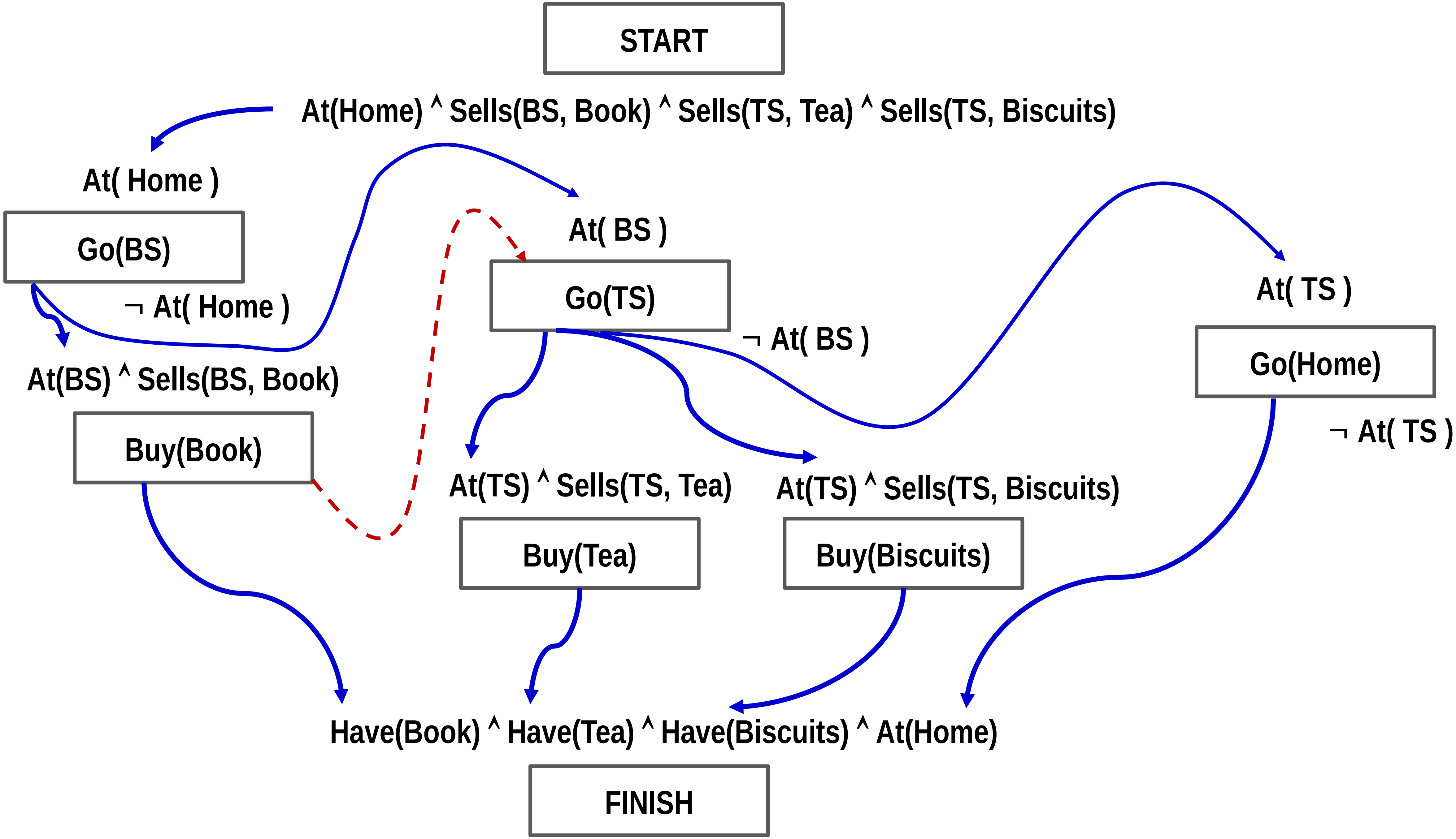
At(z)

Go(Home)

¬ At(z)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(Home)

Go(BS)

¬ At(Home)

At(BS) ^ Sells(BS, Book)

Buy(Book)

At(BS)

Go(TS)

¬ At(BS)

At(TS) ^ Sells(TS, Tea)

Buy(Tea)

At(TS) ^ Sells(TS, Biscuits)

Buy(Biscuits)

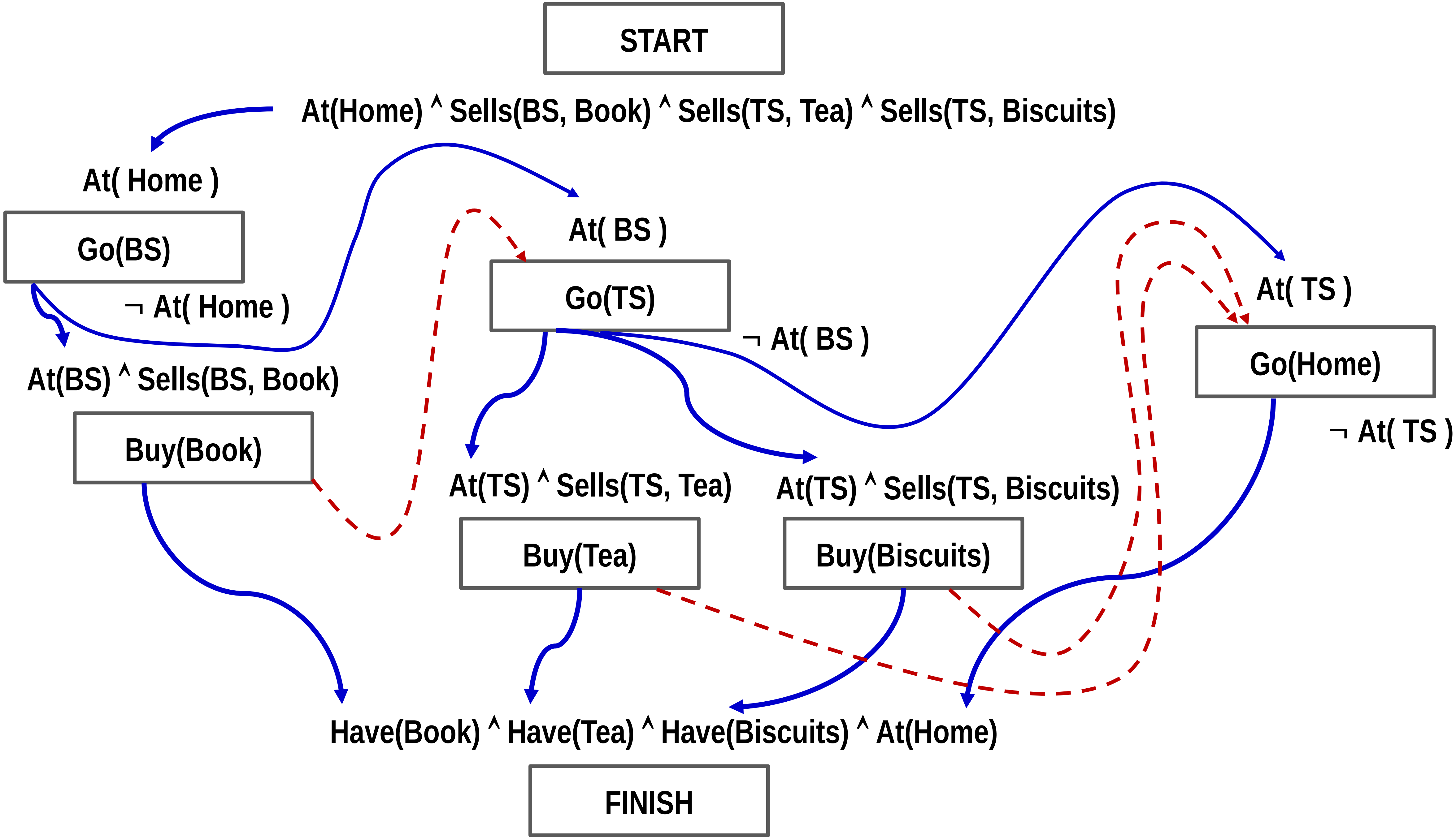
At(TS)

Go(Home)

¬ At(TS)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH



START

At(Home) ^ Sells(BS, Book) ^ Sells(TS, Tea) ^ Sells(TS, Biscuits)

At(Home)

Go(BS)

¬ At(Home)

At(BS) ^ Sells(BS, Book)

Buy(Book)

At(BS)

Go(TS)

¬ At(BS)

At(TS) ^ Sells(TS, Tea)

Buy(Tea)

At(TS) ^ Sells(TS, Biscuits)

Buy(Biscuits)

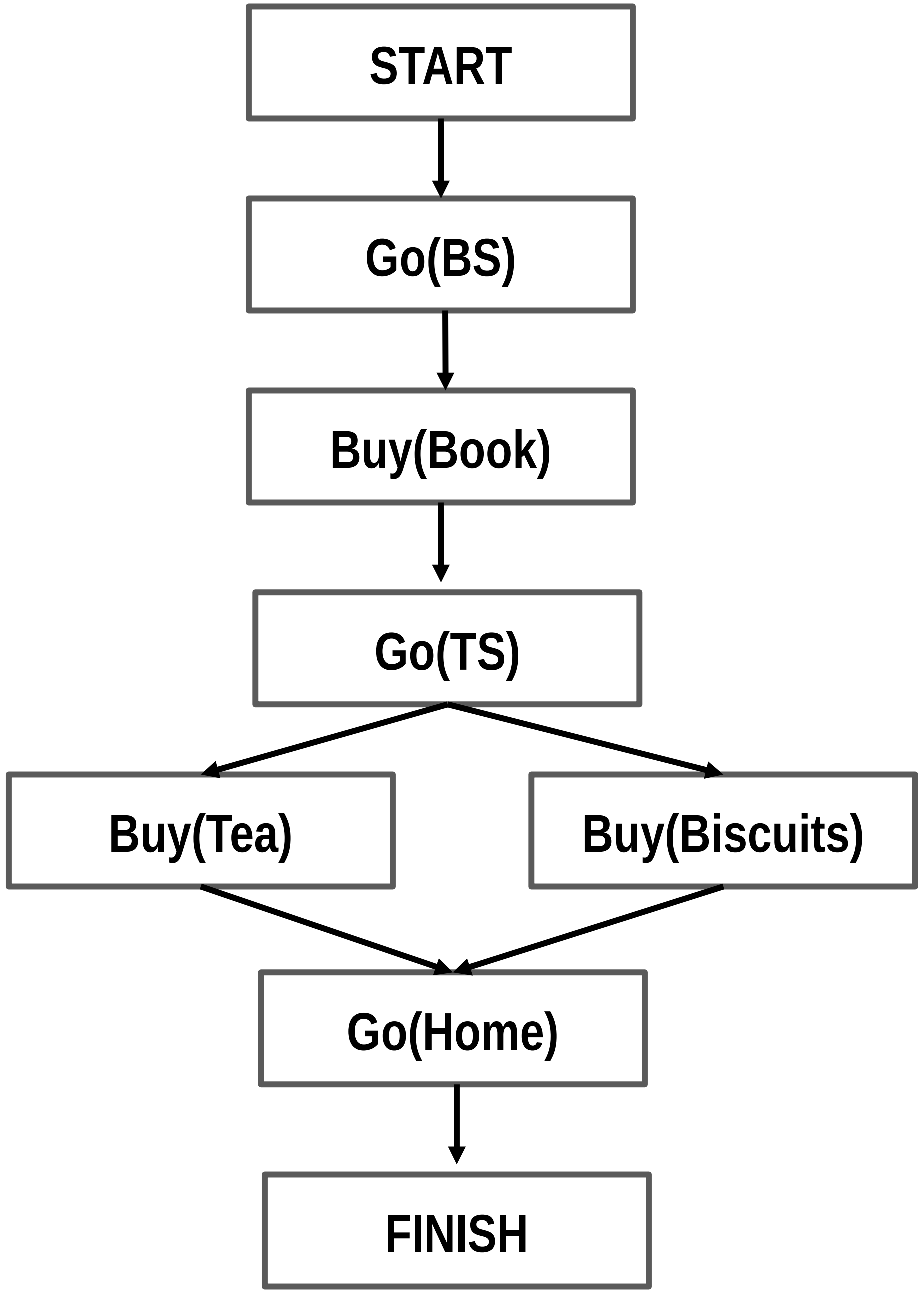
At(TS)

Go(Home)

¬ At(TS)

Have(Book) ^ Have(Tea) ^ Have(Biscuits) ^ At(Home)

FINISH


START
Go(BS)
Buy(Book)
Go(TS)
Buy(Tea)
Buy(Biscuits)
Go(Home)
FINISH

Exercise

Consider the problem of swapping the contents of two registers, A and B. For a programmer, this is very easy, but suppose we wish to ask a robot to figure out how to write such a code. Suppose we pose it as the following planning problem in STRIPS:

```
 Op( ACTION: Start,
       EFFECT: Contains(A, X) ^ Contains(B, Y))
     // Register A contains X, Register B contains Y
```

```
Op( ACTION: Finish,
       PRECOND: Contains(B, X) ^ Contains(A, Y))
```

// The following action assigns the content v1 of register r1 to register r2 which contained v2

```
Op( ACTION: Assign( r1, v1, r2, v2 ),
         PRECOND: Contains(r1, v1) ^ Contains(r2, v2),
         EFFECT: Contains(r2, v1))
```

Exercise

Consider the problem of swapping the contents of two registers, A and B. For a programmer, this is very easy, but suppose we wish to ask a robot to figure out how to write such a code. Suppose we pose it as the following planning problem in STRIPS:

Op(ACTION: Start,

EFFECT: Contains(A, X) ^ Contains(B, Y))

// Register A contains X, Register B contains Y

Op(ACTION: Finish,

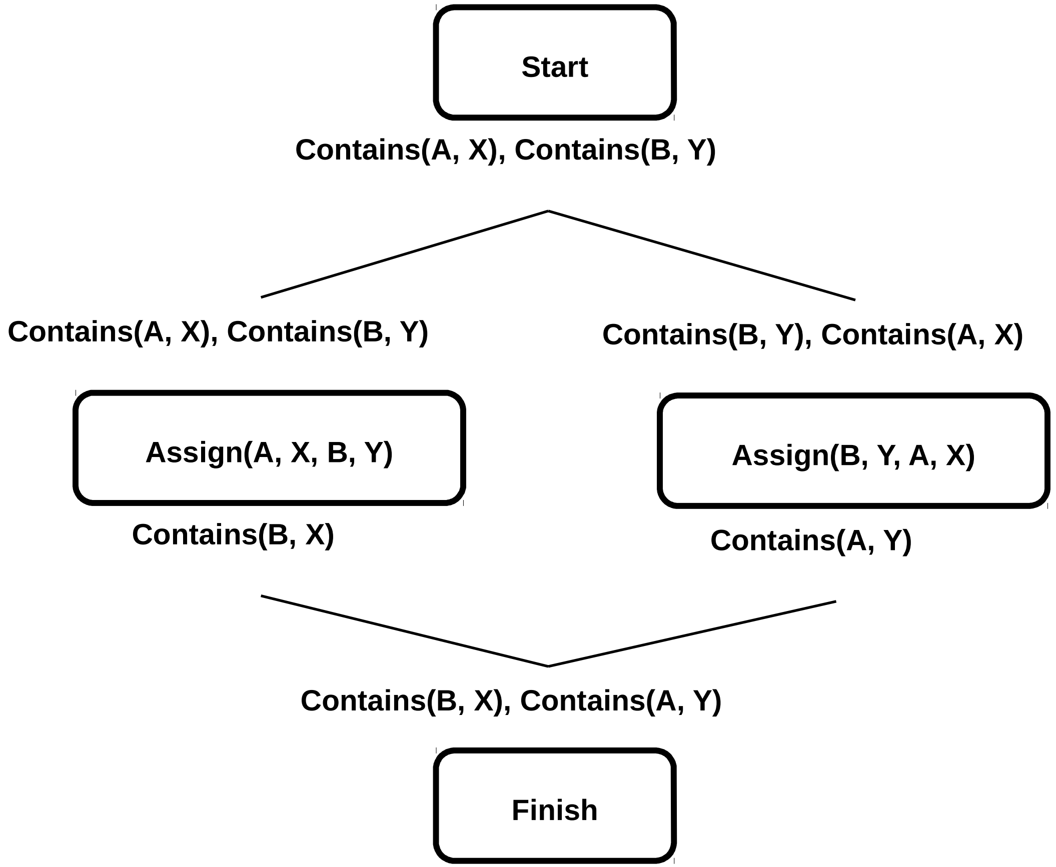
PRECOND: Contains(B, X) ^ Contains(A, Y))

// The following action assigns the content v1 of register r1 to register r2 which contained v2

Op(ACTION: Assign(r1, v1, r2, v2),

PRECOND: Contains(r1, v1) ^ Contains(r2, v2),

EFFECT: Contains(r2, v1))



Start
Contains(A, X), Contains(B, Y)
Contains(A, X), Contains(B, Y)
Contains(B, Y), Contains(A, X)
Assign(A, X, B, Y)
Assign(B, Y, A, X)
Contains(B, X)
Contains(A, Y)
Contains(B, X), Contains(A, Y)
Finish

Observe that the steps of the plan cannot be executed in any order to achieve the swapping the contents of the registers. The robot is not at fault, since it was not told that assigning the contents of register r1 to register r2 destroys the previous content of register r2. Can you rewrite the action so that the correct consequence of the action is captured?