The Wolf, Goat, and Cabbage Problem

This example shows how to use `lustre-sally` to solve a simple puzzle. This example is derived from one of the tests for `jkind`, which is another model-checker that uses Lustre as its input language.

We solve the "Wolf, goat, and cabbage problem", described on Wikipedia as follows:

> Once upon a time a farmer went to a market and purchased a wolf, a goat, and a cabbage. On his way home, the farmer came to the bank of a river and rented a boat. But crossing the river by boat, the farmer could carry only himself and a single one of his purchases: the wolf, the goat, or the cabbage.
>
> If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage.
>
> The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?

To solve the puzzle we model the problem as a *transition system*, described in Lustre. The transition system desribes the initial state of the world (e.g., the locations of the farmer and his purchases), and also what are valid ways for the world to change (e.g., constraints, such as "the goat and the wolf should not be left alone", and "the farmer may only pick up things from the side of the river that he is on").

Once we have the Lustre model we can use `lustre-sally` to invoke the `sally` model-checker, to analyze the resulting system. The model checker's job is to look for states of the world where some property holds. For the purposes of this puzzle we'll have everyone start on the left side of the river, and look for states where everyone is on the right side of the river, without anyone getting eaten in-beetween. Here is the sample output we'd like to get:

Step	`wolfLoc`	`goatLoc`	`cabbageLoc`	`farmerLoc`	`->`	`move`
1	Left	Left	Left	Left		Goat
2	Left	Right	Left	Right		Farmer
3	Left	Right	Left	Left		Cabbage
4	Left	Right	Right	Right		Goat
5	Left	Left	Right	Left		Wolf
6	Right	Left	Right	Right		Farmer
7	Right	Left	Right	Left		Goat
8	Right	Right	Right	Right		

The Lustre Model

We start by declaring a couple of enumeration types:

```
type Character  = enum { Farmer, Wolf, Goat, Cabbage };
type Side       = enum { Left, Right };
```

Type type `Character` represents the characters in the story, and the type `Side` models the characters' locations. Without loss of generality, we arbitrarily choose that the characters start on the `Left` side of the river, and are trying to get to the `Right` side.

Signature. The central part of the model is the function `action` which models a single action of the farmer:

```
node action(move : Character)
  returns (wolfLoc, goatLoc, cabbageLoc, farmerLoc : Side);
```

The function's parameter specifies who goes with the farmer in the boat, and the outputs are the new locations of the characters. If the parameter is `Farmer`, then the farmer goes alone.

Local Variables. The function uses some local state to classify the current situation:

```
var
  validMove     : bool;
  nothingEaten  : bool;
  solved        : bool;
```

The names of the variables are suggestive but next we provide precise definitions, which are needed so that the model can be analyzed.

Definition. We start with `validMove` which holds true when the farmer is following the rules, namely he may only pick up characters that are on the same bank of the river as he is:

```
let
  validMove =
    (move = Wolf    => farmerLoc = wolfLoc) and
    (move = Goat    => farmerLoc = goatLoc) and
    (move = Cabbage => farmerLoc = cabbageLoc);
```

As the name suggests, the variable `nothingEaten`, ensures that the goat and the cabbage are not eaten. We do this by insisting that if a hungry character is in the same location as their food, then the farmer is also present to ensure they behave:

```
  nothingEaten =
    (wolfLoc = goatLoc    => farmerLoc = goatLoc) and
    (goatLoc = cabbageLoc => farmerLoc = cabbageLoc);
```

The variable `solved` states precisely what we are looking for: a situation where all characters are on the `Right` river bank, and we only preformed valid moves, and no one was eaten:

```
  solved =
    wolfLoc    = Right and
    goatLoc    = Right and
    cabbageLoc = Right and
    farmerLoc  = Right and
    historically(nothingEaten and validMove);
```

Intuitively, the function `historically` says that the given property holds now, and in all previous states, and we'll see its definition shortly.

Finally, we need to specify how the characters' locations are affected by the farmer's actions:

```
    farmerLoc     = doMove(move, Farmer);
    wolfLoc       = doMove(move, Wolf);
    goatLoc       = doMove(move, Goat);
    cabbageLoc    = doMove(move, Cabbage);
```

The details of movement are factored into a separate function called `doMove`.

The final part of the specification consists of two special comments that tell the solver what we'd like to do.

```
    --%MAIN;
    --%PROPERTY not solved;
tel;
```

The *pragma* `MAIN` specifies that this is the entry point for our problem, and we are interested in the inputs and state of this function. Only one of the functions in a specification should be marked with `MAIN`. The pragma `PROPERTY` tells the solver what property we are searching for, by telling it what states to prune out. In this case, we are telling the solver to prune out states where the problem is *not* solved, thus asking it to find a solution.

While this may seems inverted, there is a good reason for it. Another way to think of `PROPERTY` is as writing an assertion: we are asking the solver to check that the property holds in all reachable states of the system. If the solver finds a state the violets the invariant, it will report it by showing a sequence of steps that lead to the problem. So, in a way, in our puzzle we are "tricking" the solver to check the "invariant" that the problem is impossible to solve, thus asking it to "prove us wrong" by finding a solution.

We still need to specify how characters are affected by the farmer's actions, which is captured in function `doMove`. It takes two characters as parameters, the first one specifies who the farmer is moving, and the second one which character's location we are computing. As a result we return the new location of the character.

```
node doMove(move, character: Character) returns (location : Side);
let
  location =
```

```
    Left ->
      if character = Farmer then
        pre(otherSide(location))
      else
      if pre(move) = character
         then pre(otherSide(location))
         else pre(location);
tel;

node otherSide(side : Side) returns (other : Side);
let
  other = if side = Left then Right else Left;
tel;
```

The function `doMove` illustrates some of Lustre's temporal combinators. Expressions of the form `A -> B` capture the common pattern where a value is initialized to `A` and after that behaves as `B`. In this case, all characters start on the `Left` bank of the river, and the expression after the arrow specifies how they are affected by the move. The other temporal operator, `pre(X)` refers to the value of `X` but in the *previous* state (e.g., where something *used* to be). Since the farmer always moves with the boat, its location alternates on every action, no matter who is being moved. The `else` statement deals with other characters and specifies that the character picked up by the farmer change their location while the other characters remain in place.

The same temporal operators are used to define `historically`, which we present without further comment, but encourage the reader to try to understand its definition:

```
node historically(x : bool) returns (holds : bool);
let
  holds = x and (true -> pre holds);
tel;
```

Running the Model

To run `lustre-sally` on the model, we just provide the file containing the specification as a parameter:

```
> lustre-sally example/farmer.lus
```

This results in the following output:

```
[Lustre] Loading model from: "example/farmer.lus"
[Lustre] Validating properties:
[Lustre] Property Prop on line 28...
  [Sally]considering 8 past states [Invalid] See (...URL...)
[Trace]
Prop on line 28:
```

```
 Step | wolfLoc | goatLoc | cabbageLoc | farmerLoc | -> | move
------|---------|---------|------------|-----------|----|---------
 1    | Left    | Left    | Left       | Left      |    | Goat
 2    | Left    | Right   | Left       | Right     |    | Farmer
 3    | Left    | Right   | Left       | Left      |    | Cabbage
 4    | Left    | Right   | Right      | Right     |    | Goat
 5    | Left    | Left    | Right      | Left      |    | Wolf
 6    | Right   | Left    | Right      | Right     |    | Farmer
 7    | Right   | Left    | Right      | Left      |    | Goat
 8    | Right   | Right   | Right      | Right     |    |
```

```
[Lustre] Summary:
  [Valid] 0
  [Unknown] 0
  [Invalid] 1
[Lustre] Model status: [Invalid]
```

While the `Invalid` markers may seem concerning, recall the discussion from the previous section: we "tricked" the solver to look for a solution by asserting that one does not exist, so the solver is simply saying that our assertion is invalid, as it was able to find a solution.

You may also note the line:

```
[Lustre] Property Prop on line 28...
  [Sally]considering 8 past states [Invalid] See (...URL...)
```

where `(...URL...)` would be some system specific location. The web-site in this URL contains the full details about the example found by the solver, in the context of the original Lustre specification. This allows for interactive exploration of the model state at each step, including the parameters and local state of the different invocations of the same function. Here is a screen shot, demonstrating some of this, but it it best to try it out!

```
type Side       = enum { Left, Right };
node action(move=Cabbage : Character)
  returns (wolfLoc=Left, goatLoc=Right, cabbageLoc=Left, farmerLoc=Left : Side);
var
  validMove=True      : bool;
  nothingEaten=True   : bool;
  solved=False         : bool;
let
  validMove =
    (move = Wolf    => farmerLoc = wolfLoc) and
    (move = Goat    => farmerLoc = goatLoc) and
    (move = Cabbage => farmerLoc = cabbageLoc);
  nothingEaten =
    (wolfLoc = goatLoc    => farmerLoc = goatLoc) and
    (goatLoc = cabbageLoc => farmerLoc = cabbageLoc);
  solved =
    wolfLoc    = Right and
    goatLoc    = Right and
    cabbageLoc = Right and
    farmerLoc  = Right and
    historically(nothingEaten and validMove);
    farmerLoc     = doMove(move, Farmer);
    wolfLoc       = doMove(move, Wolf);
    goatLoc       = doMove(move, Goat);
    cabbageLoc    = doMove(move, Cabbage);
    --%MAIN;
    --%PROPERTY not solved;
tel;
node doMove(move=Cabbage, character=Farmer : Character) returns (location=Left : Side);
let
  location =
    Left ->
      if character = Farmer then
        pre(otherSide(location))
      else
      if pre(move) = character
         then pre(otherSide(location))
         else pre(location);
tel;

node otherSide(side : Side) returns (other : Side);
let
  other = if side = Left then Right else Left;
tel;
node historically(x : bool) returns (holds : bool);
let
  holds = x and (true -> pre holds);
tel;

<< step:3 / 8 >>
```

Figure 1: Screenshot