MODULE *MissionariesAndCannibals*

This module specifies a system that models the one described in the missionaries and cannibals problem. On 20 *December* 2018, *Wikipedia* contained the following description of this problem.

> [*T*]hree missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board.

As explained below, we can use the specification and the *TLC* model checker to find a solution to the problem.

The following EXTENDS statement imports definitions of the ordinary arithmetic operations on integers and the definition of the *Cardinality* operator, where *Cardinality*(S) is the number of elements in S if S is a finite set.

EXTENDS *Integers*, *FiniteSets*

Next comes the declaration of the sets of missionaries and cannibals.

CONSTANTS *Missionaries*, *Cannibals*

In TLA+, an execution of a system is described as a sequence of states, where a state is an assignment of values to variables. A pair of successive states in an execution is called a step. We write $s \to t$ to indicate that s, t is a step in an execution.

The first thing to do when writing a spec of a system is to decide what should constitute a step. There are a number of ways to describe the cannibals and missionaries that differ in what constitutes a step. For example, we could consider a single person getting into or out of the boat to be a step. Breaking the execution into many small steps gives a more accurate description of the physical system, but using fewer big steps provides a simpler spec. We write a spec for a purpose, and we want to use the fewest steps that gives a sufficiently accurate description for that purpose. Our purpose is to find a solution to the cannibals and missionaries problem. A little thought shows that this can be done by a specification in which a step consists of moving a set of people with the boat from one bank to the other.

Having decided what a step is, we can see that a state of the system must describe on which bank the boat is and what people are on each bank. What I find to be a simple and natural way to describe that state is with the following two variables:

bank_of_boat: In any system execution, *bank_of_boat* will equal the bank of the river on which the boat is docked.

who_is_on_bank: In any execution of the system, the value of *who_is_on_bank*[b] will be the set of people on bank b.

Although we could declare a constant *Banks* to be the set of riverbanks, it's more convenient to simply give them names. Let's call them "E" (for east bank) and "W" (for west bank), so {"E", "W"} is the set of riverbanks.

VARIABLES *bank_of_boat*, *who_is_on_bank*

Although not needed to specify the system, it's a good idea to tell the reader of the spec the types of values that the variables will have in any reachable state of the system. This is conventionally done by defining a state predicate called *TypeOK*.

The value of *bank_of_boat* will be either "E" or "W" – that is, an element of the set {"E", "W"}. (The operator $\in$ means "is an element of", and is written by mathematicians as a *Greek* epsilon.)

The value of *who_is_on_bank* will be what programmers would call an array indexed by the set {"E", "W"}, and what mathematicians would call a function with domain {"E", "W"}. (Many common primitive programming languages permit only arrays with index-set/domain the set $\{0, \ldots, n\}$ for some integer n.) For each b in {"E", "W"}, the value of $who\_is\_on\_bank[b]$ will be a set of cannibals and/or missionaries – that is, an element of the set $Cannibals \cup Missionaries$, where $\cup$ is the set union operator. The expression SUBSET S is the set of all subsets of the set S, and $[D \to T]$ is the set of all arrays/functions with index-set/domain D such that $f[d]$ is an element of T for all d in D.

TLA+ allows you to write a conjunction of formulas as a list of those formulas bulleted by $\land$. A disjunction of formulas is similarly written with a list bulleted by $\lor$.

$$\begin{array}{lll} TypeOK & \triangleq & \land\ bank\_of\_boat \in \{\text{"E"}, \text{"W"}\} \\ & & \land\ who\_is\_on\_bank \in \\ & & \qquad [\{\text{"E"}, \text{"W"}\} \to \text{SUBSET}\ (Cannibals \cup Missionaries)] \end{array}$$

The possible executions of the system are specified by two formulas: an initial-state formula usually named *Init*, and a next-state formula usually named *Next*. The intial-state formula is the condition that must be true of the initial state of an execution. The next-state formula is the condition that must be true for all possible steps in an execution.

The initial-state formula *Init* asserts that the boat and all the cannibals and missionaries are on the east bank. The formula

$$[x \in D \mapsto exp(x)]$$

represents the array/function F with index-set/domain D such that $F[x]$ equals $exp(x)$ for all x in D.

$$\begin{array}{lll} Init & \triangleq & \land\ bank\_of\_boat = \text{"E"} \\ & & \land\ who\_is\_on\_bank = [i \in \{\text{"E"}, \text{"W"}\} \mapsto \\ & & \qquad \text{IF}\ i = \text{"E"}\ \text{THEN}\ Cannibals \cup Missionaries \\ & & \qquad\quad \text{ELSE}\ \{\}] \end{array}$$

We now define some operators that will be used to define the next-state formula *Next*.

We first define $IsSafe(S)$ to be the condition for it to be safe for S to be the set of people on a bank of the river. It is true iff there are either no missionaries in S or the cannibals in do not outnumber the missionaries in S. The operator $\subseteq$ is the subset relation, and $\cap$ is the set intersection operator.

$$\begin{array}{lll} IsSafe(S) & \triangleq & \lor\ S \subseteq Cannibals \\ & & \lor\ Cardinality(S \cap Cannibals) \leq Cardinality(S \cap Missionaries) \end{array}$$

We define *OtherBank* so that $OtherBank(\text{"E"})$ equals "W" and $OtherBank(\text{"W"})$ equals "E".

$$OtherBank(b) \triangleq \text{IF}\ b = \text{"E"}\ \text{THEN}\ \text{"W"}\ \text{ELSE}\ \text{"E"}$$

We now define the formula $Move(S, b)$ to describe a step $s \to t$ that represents a safe move of a set S of people from riverbank b to riverbank $OtherBank(b)$ – that is, a step where state t is one in which the set of people on each bank is safe. Formula $Move(S)$ contains primed and unprimed variables, where an unprimed variable v equals the value of v in state s and a primed variable v' equals the variable's value in state t. The possible step $s \to t$ describes a safe move of the people in S from b to $OtherBank(b)$ if and only if $Move(S, b)$ equals true for that step.

The definition uses the TLA+ LET /IN construct for introducing definitions local to an expression, where LET *defs* IN *exp* is the expression *exp* in which each identifier defined in *defs* has its indicated meaning. In this definition, *newThisBank* and *newOtherBank* are defined locally to equal the sets of people on bank b and on bank
OtherBank(b) after the set S of people take the boat from b to
OtherBank(b). The operator $\setminus$ is set difference, where $T \setminus S$ is the set of all elements in T not in S.

Observe that the first two conjuncts in the IN expression contain no prime variables. They are enabling conditions – conditions on state s that allow the step. The second two conjuncts specify the new values of the two variables (their values in state t) in terms of their old values (their values in state s).

$$
\begin{array}{l}
Move(S,\, b) \triangleq \; \land\, Cardinality(S) \in \{1,\, 2\} \\
\qquad\qquad \land \text{LET } newThisBank \triangleq who\_is\_on\_bank[b] \setminus S \\
\qquad\qquad\qquad\qquad\; newOtherBank \triangleq who\_is\_on\_bank[OtherBank(b)] \cup S \\
\qquad\qquad\; \text{IN} \quad \land\, IsSafe(newThisBank) \\
\qquad\qquad\qquad\quad \land\, IsSafe(newOtherBank) \\
\qquad\qquad\qquad\quad \land\, bank\_of\_boat' = OtherBank(b) \\
\qquad\qquad\qquad\quad \land\, who\_is\_on\_bank' = \\
\qquad\qquad\qquad\qquad [i \in \{\text{“E”},\ \text{“W”}\} \mapsto \text{IF } i = b \text{ THEN } newThisBank \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } newOtherBank]
\end{array}
$$

The next-state formula *Next* describes all steps $s \to t$ that represent a safe move of a set S of people across the river starting from *bank_of_boat*. It asserts that there exists some subset S of the set of people on the bank where the boat is for which step $s \to t$ describes a safe movement of the people in S to the other bank. This assertion is expressed mathematically with the existential quantification operator $\exists$ (written by mathematicians as an upside down E), where

$$\exists\, x \in T : A(x)$$

asserts that $A(x)$ is true for at least one value x in the set T.

$$
\begin{array}{l}
Next \triangleq \exists\, S \in \text{SUBSET } who\_is\_on\_bank[bank\_of\_boat] : \\
\qquad\qquad Move(S,\, bank\_of\_boat)
\end{array}
$$

The usual reason for writing a spec is to check the system you're specifying for errors. This means checking that all possible executions satisfy some property. The most commonly checked property is invariance, asserting that some condition is satisfied by every state in in every possible execution.

The purpose of this spec is to solve the cannibals and missionaries problem, which means finding some possible execution in which everyone reaches bank “W”. We can find that solution by having the *TLC* model checker check the invariance property that, in every reachable state, there is someone left on bank “E”. When *TLC* find that an invariant it's checking isn't an invariant, it outputs an execution that reaches a state in which the invariant isn't true–which in this case means an execution that solves the problem (one ending in a state with no one on bank “E”). So to find the solution, you just have to run *TLC* on a model of this specification in which three-element sets are substituted for the constants *Missionaries* and *Cannibals*, instructing *TLC* to check that the formula

$$who\_is\_on\_bank[\text{“E”}] \neq \{\}$$

is an invariant. The error trace *TLC* produces is a solution to the problem. You can run *TLC* from the TLA+ *Toolbox*. Go to the TLA+ web page to find out how to learn to do that.

This problem was proposed to me by Jay *Misra*, who then suggested improvements to my first version of the spec.

* Modification History
* Last modified Sat *Dec* 22 14:17:18 *PST* 2018 by *lamport*
* Created *Thu Dec* 20 11:44:08 *PST* 2018 by *lamport*