MODULE *Consensus*

The consensus problem requires a set of processes to choose a single value. This module specifies the problem by specifying exactly what the requirements are for choosing a value.

EXTENDS *Naturals*, *FiniteSets*, *FiniteSetTheorems*, *TLAPS*

We let the constant parameter *Value* be the set of all values that can be chosen.

CONSTANT *Value*

****

We now specify the safety property of consensus as a trivial algorithm that describes the allowed behaviors of a consensus algorithm. It uses the variable *chosen* to represent the set of all chosen values. The algorithm is trivial; it allows only behaviors that contain a single state-change in which the variable *chosen* is changed from its initial value $\{\}$ to the value $\{v\}$ for an arbitrary value v in *Value*. The algorithm itself does not specify any fairness properties, so it also allows a behavior in which *chosen* is not changed. We could use a translator option to have the translation include a fairness requirement, but we don't bother because it is easy enough to add it by hand to the safety specification that the translator produces.

A real specification of consensus would also include additional variables and actions. In particular, it would have Propose actions in which clients propose values and Learn actions in which clients learn what value has been chosen. It would allow only a proposed value to be chosen. However, the interesting part of a consensus algorithm is the choosing of a single value. We therefore restrict our attention to that aspect of consensus algorithms. In practice, given the algorithm for choosing a value, it is obvious how to implement the Propose and Learn actions.

For convenience, we define the macro *Choose*() that describes the action of changing the value of *chosen* from $\{\}$ to $\{v\}$, for a nondeterministically chosen v in the set *Value*. (There is little reason to encapsulate such a simple action in a macro; however our other specs are easier to read when written with such macros, so we start using them now.) The *when* statement can be executed only when its condition, $chosen = \{\}$, is true. Hence, at most one *Choose*() action can be performed in any execution. The *with* statement executes its body for a nondeterministically chosen v in *Value*. Execution of this statement is enabled only if *Value* is non-empty–something we do not assume at this point because it is not required for the safety part of consensus, which is satisfied if no value is chosen.

We put the *Choose*() action inside a *while* statement that loops forever. Of course, only a single *Choose*() action can be executed. The algorithm stops after executing a *Choose*() action. Technically, the algorithm deadlocks after executing a *Choose*() action because control is at a statement whose execution is never enabled. Formally, termination is simply deadlock that we want to happen. We could just as well have omitted the *while* and let the algorithm terminate. However, adding the *while* loop makes the TLA+ representation of the algorithm a tiny bit simpler.

```
--algorithm Consensus{
  variable chosen = {} ;
  macro Choose( ) { when chosen = {} ;
                    with ( v \in Value ) { chosen := {v} }  }
   { lbl: while ( TRUE ) { Choose() }
    }
 }
```

The *PlusCal* translator writes the TLA+ translation of this algorithm below. The formula *Spec* is the TLA+ specification described by the algorithm's code. For now, you should just understand its two subformulas *Init* and *Next*. Formula *Init* is the initial predicate and describes all possible initial states of an execution. Formula *Next* is the next-state relation; it describes the possible state changes (changes of the values of variables), where unprimed variables represent their values in the old state and primed variables represent their values in the new state.

****

**** BEGIN TRANSLATION

VARIABLE *chosen*

$$vars \triangleq \langle chosen \rangle$$

$$Init \triangleq$$

Global variables

$$\land\ chosen = \{\}$$

$$\begin{aligned} Next \triangleq\ & \land\ chosen = \{\} \\ & \land\ \exists\, v \in Value : \\ & \qquad chosen' = \{v\} \end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

**** END TRANSLATION

We now prove the safety property that at most one value is chosen. We first define the type-correctness invariant *TypeOK*, and then define *Inv* to be the inductive invariant that asserts *TypeOK* and that the cardinality of the set *chosen* is at most 1. We then prove that, in any behavior satisfying the safety specification *Spec*, the invariant *Inv* is true in all states. This means that at most one value is chosen in any behavior.

$$\begin{aligned} TypeOK \triangleq\ & \land\ chosen \subseteq Value \\ & \land\ IsFiniteSet(chosen) \end{aligned}$$

$$\begin{aligned} Inv \triangleq\ & \land\ TypeOK \\ & \land\ Cardinality(chosen) \leq 1 \end{aligned}$$

We now prove that *Inv* is an invariant, meaning that it is true in every state in every behavior. Before trying to prove it, we should first use *TLC* to check that it is true. It's hardly worth bothering to either check or prove the obvious fact that *Inv* is an invariant, but it's a nice tiny exercise. Model checking is instantaneous when *Value* is set to any small finite set.

To understand the following proof, you need to understand the formula *Spec* , which equals

$Init \land \Box[Next]\_vars$

where *vars* is the tuple $\langle chosen, pc \rangle$ of all variables. It is a temporal formula satisfied by a behavior iff the behavior starts in a state satisfying *Init* and such that each step (sequence of states) satisfies $[Next]\_vars$, which equals

$Next \lor\ (vars' = vars)$

Thus, each step satisfies either *Next* (so it is a step allowed by the next-state relation) or it is a "stuttering step" that leaves all the variables unchanged. The reason why a spec must allow stuttering steps will become apparent when we prove that a consensus algorithm satisfies this specification of consensus.

The following lemma asserts that *Inv* is an inductive invariant of the next-state action *Next*. It is the key step in proving that *Inv* is an invariant of (true in every behavior allowed by) specification *Spec*.

LEMMA $InductiveInvariance \triangleq$

$$Inv \land [Next]_{vars} \Rightarrow Inv'$$

$\langle 1 \rangle$.SUFFICES ASSUME Inv, $[Next]_{vars}$
PROVE Inv'

OBVIOUS

$\langle 1 \rangle$1.CASE $Next$

In the following BY proof, $\langle 1 \rangle 1$ denotes the case assumption *Next*

BY $\langle 1 \rangle 1$, $FS\_EmptySet$, $FS\_AddElement$ DEF Inv, $TypeOK$, $Next$

$\langle 1 \rangle$2.CASE $vars' = vars$

BY $\langle 1 \rangle 2$ DEF Inv, $TypeOK$, $vars$

$\langle 1 \rangle$3. QED

BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$ DEF $Next$

THEOREM $Invariance \triangleq Spec \Rightarrow \Box Inv$

$\langle 1 \rangle$1. $Init \Rightarrow Inv$

BY $FS\_EmptySet$ DEF $Init$, Inv, $TypeOK$

$\langle 1 \rangle$2. QED

BY PTL, $\langle 1 \rangle 1$, $InductiveInvariance$ DEF $Spec$

We now define *LiveSpec* to be the algorithm's specification with the added fairness condition of weak fairness of the next-state relation, which asserts that execution does not stop if some action is enabled. The temporal formula *Success* asserts that some value is eventually chosen. Below, we prove that *LiveSpec* implies *Success* This means that, in every behavior satisfying *LiveSpec*, some value is chosen.

$LiveSpec \triangleq Spec \land \mathrm{WF}_{vars}(Next)$
$Success \triangleq \Diamond(chosen \neq \{\})$

For liveness, we need to assume that there exists at least one value.

ASSUME $ValueNonempty \triangleq Value \neq \{\}$

TLAPS does not yet reason about ENABLED . Therefore, we must omit all proofs that involve ENABLED formulas. To perform as much of the proof as possible, as much as possible we restrict the use of an ENABLED expression to a step asserting that it equals its definition. ENABLED A is true of a state s iff there is a state t such that the step $s \rightarrow t$ satisfies A. It follows from this semantic definition that ENABLED A equals the formula obtained by

1. Expanding all definitions of defined symbols in A until all primes are priming variables.
2. For each primed variable, replacing every instance of that primed variable by a new symbol (the same symbol for each primed variable).
3. Existentially quantifying over those new symbols.

LEMMA $EnabledDef \triangleq$
$TypeOK \Rightarrow$
$((\text{ENABLED } \langle Next \rangle_{vars}) \equiv (chosen = \{\}))$

$\langle 1 \rangle$ DEFINE $E \triangleq$

$$\exists\, chosenp : \begin{array}{l} \land \begin{array}{l} \land\, chosen = \{\} \\ \land\, \exists\, v \in Value : chosenp = \{v\} \end{array} \\ \land\, \neg(\langle chosenp \rangle = \langle chosen \rangle) \end{array}$$

$\langle 1 \rangle 1.\ E = \text{ENABLED}\ \langle \mathit{Next} \rangle_{\mathit{vars}}$

BY DEF *Next*, *vars*

PROOF OMITTED

$\langle 1 \rangle 2.$ SUFFICES ASSUME TypeOK

PROVE $E = (\mathit{chosen} = \{\})$

BY $\langle 1 \rangle 1$, *Zenon*

$\langle 1 \rangle 3.\ E = \exists\, \mathit{chosenp} : E!(\mathit{chosenp})!1$

BY $\langle 1 \rangle 2$, *Isa* DEF *TypeOK*

$\langle 1 \rangle 4.\ @ = (\mathit{chosen} = \{\})$

BY $\langle 1 \rangle 2$, *ValueNonempty*, *Zenon* DEF *TypeOK*

$\langle 1 \rangle 5.$ QED

BY $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, *Zenon*

Here is our proof that *Livespec* implies *Success*. It uses the standard TLA proof rules. For example *RuleWF*1 is defined in the *TLAPS* module to be the rule *WF*1 discussed in

```
AUTHOR  = "Leslie Lamport",
TITLE   = "The Temporal Logic of Actions",
JOURNAL = toplas,
volume  = 16,
number  = 3,
YEAR    = 1994,
month   = may,
PAGES   = "872--923"
```

PTL stands for propositional temporal logic reasoning. We expect that, when *TLAPS* handles temporal reasoning, it will use a decision procedure for *PTL*.

THEOREM $\mathit{LiveSpec} \Rightarrow \mathit{Success}$

$\langle 1 \rangle 1. \Box \mathit{Inv} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathrm{WF}_{\mathit{vars}}(\mathit{Next}) \Rightarrow (\mathit{chosen} = \{\} \leadsto \mathit{chosen} \neq \{\})$

$\langle 2 \rangle.$DEFINE $P \triangleq \mathit{chosen} = \{\}$

$Q \triangleq \mathit{chosen} \neq \{\}$

$\langle 2 \rangle 1.$ SUFFICES $\Box[\mathit{Next}]_{\mathit{vars}} \land \mathrm{WF}_{\mathit{vars}}(\mathit{Next}) \Rightarrow ((\mathit{Inv} \land P) \leadsto Q)$

BY *PTL*

$\langle 2 \rangle 2.\ (\mathit{Inv} \land P) \land [\mathit{Next}]_{\mathit{vars}} \Rightarrow ((\mathit{Inv}' \land P') \lor Q')$

BY *InductiveInvariance*

$\langle 2 \rangle 3.\ (\mathit{Inv} \land P) \land \langle \mathit{Next} \rangle_{\mathit{vars}} \Rightarrow Q'$

BY DEF *Inv*, *Next*, *vars*

$\langle 2 \rangle 4.\ (\mathit{Inv} \land P) \Rightarrow \text{ENABLED}\ \langle \mathit{Next} \rangle_{\mathit{vars}}$

BY *EnabledDef* DEF *Inv*

$\langle 2 \rangle.$HIDE DEF P, Q

$\langle 2 \rangle.$QED

BY $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, $\langle 2 \rangle 4$, *PTL*

$\langle 1 \rangle 2.\ (\mathit{chosen} = \{\} \leadsto \mathit{chosen} \neq \{\}) \Rightarrow ((\mathit{chosen} = \{\}) \Rightarrow \Diamond(\mathit{chosen} \neq \{\}))$

BY *PTL*

$\langle 1 \rangle 3.$ QED

BY *Invariance*, $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, *PTL* DEF *LiveSpec*, *Spec*, *Init*, *Success*

The following theorem is used in the refinement proof in module *VoteProof*.

THEOREM $LiveSpecEquals \triangleq$

$$LiveSpec \equiv Spec \land (\Box\Diamond\langle Next\rangle_{vars} \lor \Box\Diamond(chosen \neq \{\}))$$

$\langle 1\rangle 1.\ \land Spec \equiv Spec \land \Box TypeOK$

$\quad\ \land LiveSpec \equiv LiveSpec \land \Box TypeOK$

BY $Invariance$, PTL DEF $LiveSpec$, Inv

$\langle 1\rangle 2.\ (chosen \neq \{\}) \equiv \neg(chosen = \{\})$

OBVIOUS

$\langle 1\rangle 3. \Box TypeOK \Rightarrow ((\Box\Diamond\neg \text{ENABLED } \langle Next\rangle_{vars}) \equiv \Box\Diamond(chosen \neq \{\}))$

BY $\langle 1\rangle 2$, $EnabledDef$, PTL

$\langle 1\rangle 4.$ QED

BY $\langle 1\rangle 1$, $\langle 1\rangle 3$, PTL DEF $LiveSpec$

* Modification History
* Last modified *Mon* May 11 18:36:27 *CEST* 2020 by *merz*
* Last modified *Mon Aug* 18 15:00:45 *CEST* 2014 by *tomer*
* Last modified *Mon Aug* 18 14:58:57 *CEST* 2014 by *tomer*
* Last modified *Tue Feb* 14 13:35:49 *PST* 2012 by *lamport*
* Last modified *Mon Feb* 07 14:46:59 *PST* 2011 by *lamport*