LOGICAL MODELS OF SYSTEMS

Theory and Software

Wolfgang Schreiner <Wolfgang.Schreiner@risc.jku.at>
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

JOHANNES KEPLER
UNIVERSITY LINZ

Logical Models

What is the purpose of logical modeling?

- Precisely describe the problem to be solved.
  - Clarification of mind, resolution of ambiguities.
  - Specification of program to be developed.
- Software-supported analysis of the problem and its solution.
  - Validation of specification.
  - Validation/verification of solution.
  - Interactive/automatic provers and model checkers.
- Automatic computation of solution respectively simulation of execution.
  - Logical solvers (SMT: Satisfiability Modulo Theories).
  - Perhaps: rapid prototyping of a later manually written program.

To profit from software, we need computer-understandable models.

1. **Modeling Systems**

2. **The Temporal Logic of Actions (TLA)**

Computational Systems

Programs are just special cases of "(computational) systems".

- Computational System
  - One or more active components.
  - Deterministic or nondeterministic behavior.
  - May or may not terminate.
- Safety
  - "Nothing bad will ever happen."
  - Partial correctness of programs: for every admissible input, if the program terminates, its output does not violate the output condition.
- Liveness
  - "Something good will eventually happen."
  - Termination of programs: for every input, the program eventually terminates.

General goal is to establish the safety and liveness of computational systems.

Transition Systems

Any computational system can be modeled as a transition system $T = (S, I, R)$.

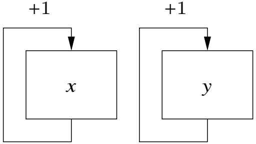
- State space $S = S_1 \times \ldots \times S_n$: the set of all possible system states.
  - Determined by the possible values of system variables $x_1, \ldots, x_n$ with values from (finite or infinite) domains $S_1, \ldots, S_n$.
- Initial states $I \subseteq S$: the possible starts of the execution of the system.
  - Typically defined by an a predicate I_x on the system variables $x_1, \ldots, x_n$.
- Transition relation $R \subseteq S \times S$: the possible execution steps.
  - Typically defined by a predicate $R_{x,x'}$ between the prestate values x and the poststate values x' of the program variables.

Nondeterminism: for some prestate x there may be multiple poststates x'.

Example

System $C = (S, I, R)$ with counters x und y which may be independently incremented.

$$S := \mathbb{Z} \times \mathbb{Z}$$
$$I(x, y) :\Leftrightarrow x = y \wedge y \geq 0$$
$$R(\langle x, y\rangle, \langle x', y'\rangle) :\Leftrightarrow$$
$$(x' = x + 1 \wedge y' = y) \vee$$
$$(x' = x \wedge y' = y + 1)$$



- Infinitely many starting states.

$$[x = 0, y = 0], [x = 1, y = 1], [x = 2, y = 2], \ldots$$

- In each state two possibilities.

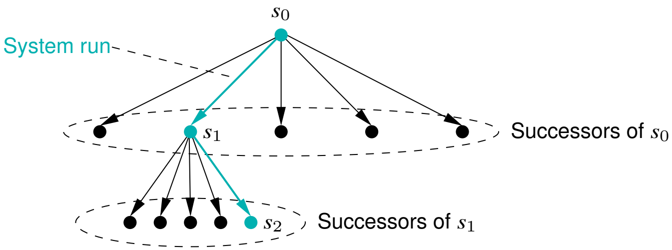
$$[x = 2, y = 3] \rightarrow [x = 3, y = 3]$$
$$\rightarrow [x = 2, y = 4]$$

A nondeterministic system.

System Runs

Transition system $T = (S, I, R)$.

- System run: (finite or infinite) sequence $s_0 \to s_1 \to s_2 \to \ldots$ of states in S.
  - s_0 is initial: $I(s_0)$.
  - $s_i \to s_{i+1}$ ist a transition: $R(s_0, s_1)$.
  - If run stops in s_n, then s_n has no successor: $\neg R(s_n, s')$, for all $s' \in S$.

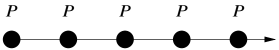


System runs can be understood as paths in a directed graph.

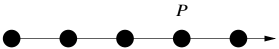
System Properties

Properties of a transition system can be specified in linear temporal logic (LTL).

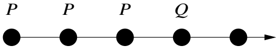
- System S satisfies LTL formula P, if each possible run of S satisfies P.
- Action: A
  - Classical logic formulas with variables $x, y, \ldots$ and $x', y', \ldots$.
  - First state pair (s_0, s_1) of run satisfies A with $x, y, \ldots$ interpreted in s_0 and $x', y', \ldots$ interpreted in s_1.
- Always: $\Box P$



  - Run satisfies property P from every position i on.
- Eventually: $\Diamond P$



  - Run satisfies P from at least one position i on.
- Until: P U Q



  - Run satisfies property Q from at least one position i on; from all previous positions $j < i$ it satisfies property P.

Example

System $C = (S, I, R)$.

$$S := \mathbb{Z} \times \mathbb{Z}$$

$$I(x, y) :\Leftrightarrow x = y \wedge y \geq 0$$

$$R(\langle x, y\rangle, \langle x', y'\rangle) :\Leftrightarrow$$
$$(x' = x + 1 \wedge y' = y) \vee$$
$$(x' = x \wedge y' = y + 1)$$

- Safety: $\Box(x \geq 0 \wedge y \geq 0)$
  - Both x and y never become negative.
  - System satisfies specification, because every run has this property.
- Liveness: $\Diamond x \geq 1$.
  - Variable x will eventually have a value greater equal 1.
  - System violates specification, because one run does not have this property:

$$[x = 0, y = 0] \to [x = 0, y = 1] \to [x = 0, y = 2] \to [x = 0, y = 3] \to \ldots$$

Liveness properties may be violated by *unfair* runs; we want to ignore such runs.

Verifying Safety

We only consider the verification of a safety property.

- $M \models \Box F$.
  - Verify that formula F is an invariant of system M.
- $M = (S, I, R)$.
  - $I(s) :\Leftrightarrow \ldots$
  - $R(s, s') :\Leftrightarrow R_0(s, s') \vee R_1(s, s') \vee \ldots \vee R_{n-1}(s, s')$.
- Proof by induction.
  - $\forall s.\ I(s) \Rightarrow F(s)$.
    - F holds in every initial state.
  - $\forall s, s'.\ F(s) \wedge R(s, s') \Rightarrow F(s')$.
    - Each transition preserves F.
    - Reduces to a number of subproofs:

$$F(s) \wedge R_0(s, s') \Rightarrow F(s')$$
$$\ldots$$
$$F(s) \wedge R_{n-1}(s, s') \Rightarrow F(s')$$

Fairness

- Infinity: Infinite $P :\Leftrightarrow \Box\Diamond P$
  - For every position i there is a position $j \geq i$ at which property P holds.
  - Property P is satisfied infinitely often.
- Stability: Stable $P :\Leftrightarrow \Diamond\Box P$
  - There is a position i such that at all positions $j \geq i$ property P holds.
  - Property P eventually permanently holds.
- Executability: Enabled A
  - Action A describes a transition that is executable in the current state s: there is a state s' with $R(s, s')$ such that $A(s, s')$.
- Weak Fairness: WF $A :\Leftrightarrow$ Stable (Enabled A) $\Rightarrow$ Infinite A
  - If A is eventually permanently enabled, then A will (infinitely often) be executed.
- Strong Fairness: SF $A :\Leftrightarrow$ Infinite (Enabled A) $\Rightarrow$ Infinite A
  - If A is infinitely often enabled, then A will (infinitely often) be executed.

Example

System $C = (S, I, R)$.

$$
\begin{aligned}
&S := \mathbb{Z} \times \mathbb{Z} \\
&I(x, y) :\Leftrightarrow x = y \wedge y \geq 0 \\
&R(\langle x, y \rangle, \langle x', y' \rangle) :\Leftrightarrow \\
&\quad (x' = x + 1 \wedge y' = y) \vee \\
&\quad (x' = x \wedge y' = y + 1)
\end{aligned}
$$

- Liveness under the Assumption of Weak Fairness:

$$(\text{WF } x' = x + 1 \wedge y' = y) \Rightarrow \Diamond x \geq 1$$

  - If first action is eventually permanently enabled, it is infinitely often executed.
  - The action is always enabled (Enabled $x' = x + 1 \wedge y' = y \equiv true$).
  - Thus it is infinitely often executed such that eventually $x \geq 1$ holds ($\Diamond x \geq 1$).

The process scheduler must implement the required fairness properties.

1. Modeling Systems

2. The Temporal Logic of Actions (TLA)

The Temporal Logic of Actions (TLA)

- Leslie Lamport (Microsoft Research since 2001).
  - ACM Turing Award 2013.
- TLA model of a system:

$$I_x \wedge \Box[R]_x \wedge \mathrm{WF}_x(A) \wedge \ldots$$

  - Initial condition I_x.
  - Transition relation $[R]_x$:
    - $[R]_x \equiv (R \vee x = x')$
    - $x = x'$: stutter step (nothing changes).
  - Fairness conditions:
    - Conjunction of formulas $\mathrm{WF}_x(A)$ and/or $\mathrm{SF}_x(A)$ for actions A.

http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

Example

$$
\begin{aligned}
X \equiv\ & \land\ x' = x + 1 \\
& \land\ y' = y
\end{aligned}
$$

$$
\begin{aligned}
Y \equiv\ & \land\ y' = y + 1 \\
& \land\ x' = x
\end{aligned}
$$

$$
\begin{aligned}
S \equiv\ & \land\ (x = 0) \land (y = 0) \\
& \land\ \Box[X \lor Y]_{\langle x,y \rangle} \\
& \land\ \mathsf{WF}_{\langle x,y \rangle}(X) \land \mathsf{WF}_{\langle x,y \rangle}(Y)
\end{aligned}
$$

$$[x = 0, x = 0] \longrightarrow [x = 1, y = 0] \longrightarrow [x = 1, y = 0] \longrightarrow [x = 1, y = 1] \longrightarrow \ldots$$

System is described in a structured way by the logical composition of actions.

TLA+

TLA is not just a logic.

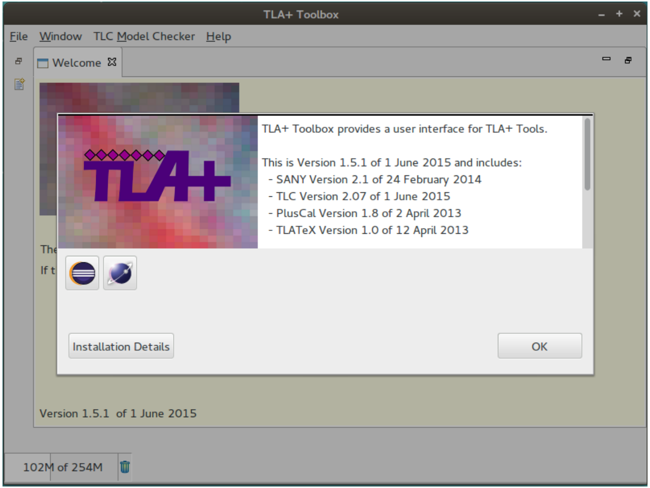
- TLA+: A formal specification language based on TLA.
  - Values from the theory of sets (no static type system).

    Chris Newcombe et al. *How Amazon Web Services Uses Formal Methods.* Communications of the ACM, vol. 58 no. 4, pages 66-73, April 2015. https://doi.org/10.1145/2699417

- TLA+ Toolbox: an IDE for various TLA tools.
  - Writing and syntax checking of TLA+ specifications.
  - Pretty printer for generation of LaTeXdocuments.
  - Translator from the algorithmic language PlusCal to TLA+.
  - Simulation and model checking of TLA+-specifications.
  - Derivation and checking of TLA+ proofs.

http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html

TLA+ Toolbox


TLA+ Toolbox
File Window TLC Model Checker Help
Welcome
TLA+
TLA+ Toolbox provides a user interface for TLA+ Tools.
This is Version 1.5.1 of 1 June 2015 and includes:
- SANY Version 2.1 of 24 February 2014
- TLC Version 2.07 of 1 June 2015
- PlusCal Version 1.8 of 2 April 2013
- TLATeX Version 1.0 of 12 April 2013
Installation Details
OK
Version 1.5.1 of 1 June 2015
102M of 254M

Example (Plain Text)

```
--------------- MODULE Counter ---------------
EXTENDS Naturals
VARIABLE x,y

I == x = 0 /\ y = 0 (* the initial state condition *)

X == /\ x' = x+1    (* increment x *)
     /\ y' = y
Y == /\ x' = x      (* increment y *)
     /\ y' = y+1
R == \/ X           (* increment x or y *)
     \/ Y

var == «x,y»        (* the system variables *)

C == I /\ [][R]_var /\ WF_var(X) /\ WF_var(Y) (* the whole specification *)

NotNegative == [](x >= 0 /\ y >= 0)          (* some properties *)
BecomeOne   == <>(x = 1 /\ y = 1)
=============================================================================
```

Example (LaTeX)

MODULE *Counter*

EXTENDS *Naturals*

VARIABLE $x,\ y$

the initial state condition

$$I \triangleq x = 0 \land y = 0$$

$$X \triangleq \land\ x' = x + 1 \quad \text{increment } x$$
$$\land\ y' = y$$

$$Y \triangleq \land\ x' = x \quad \text{increment } y$$
$$\land\ y' = y + 1$$

$$R \triangleq \lor\ X \quad \text{increment } x \text{ or } y$$
$$\lor\ Y$$

$$var \triangleq \langle x,\ y \rangle \quad \text{the system variables}$$

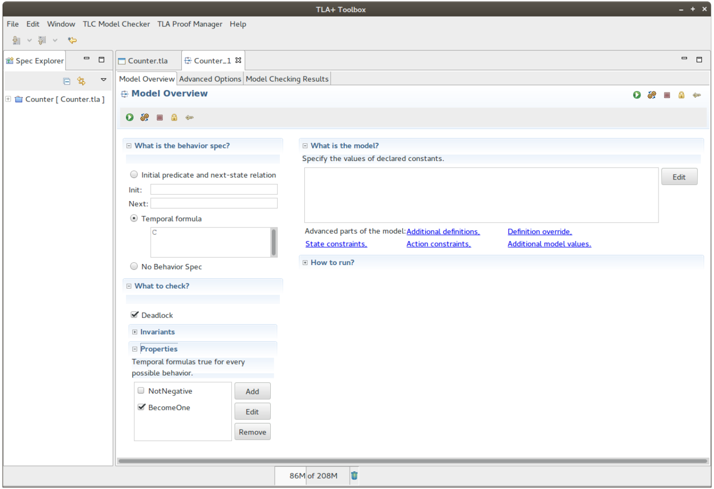
the whole specification

$$C \triangleq I \land \Box[R]_{var} \land \mathrm{WF}_{var}(X) \land \mathrm{WF}_{var}(Y)$$

some properties

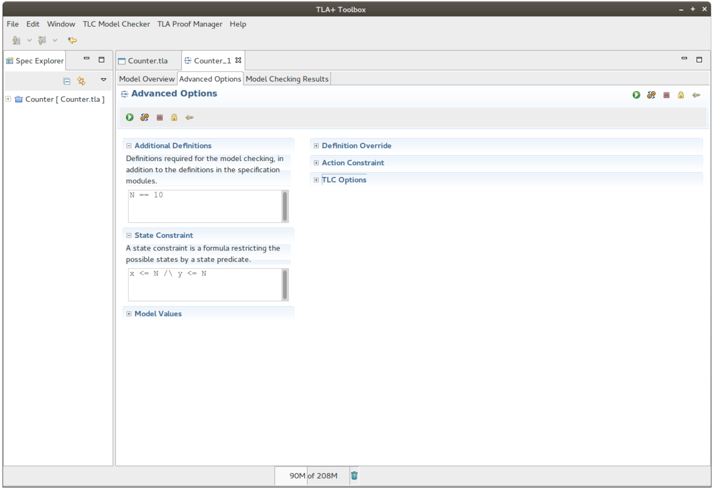
$$NotNegative \triangleq \Box(x \geq 0 \land y \geq 0)$$
$$BecomeOne \triangleq \Diamond(x = 1 \land y = 1)$$

The TLC Model Checker



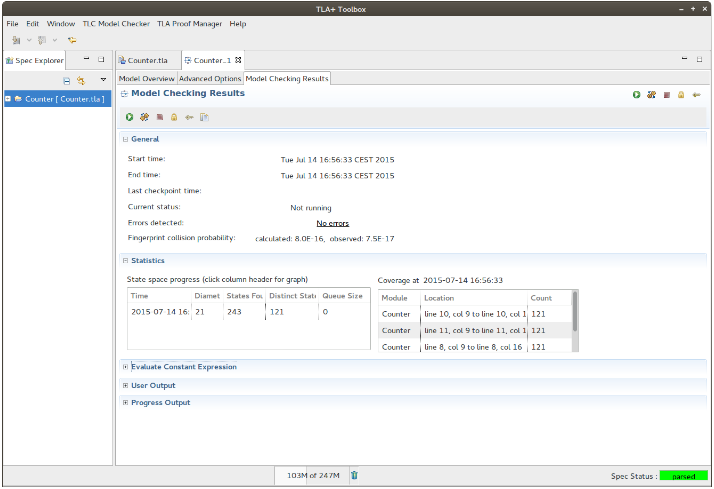
Select specification and properties to be checked.

The TLC Model Checker



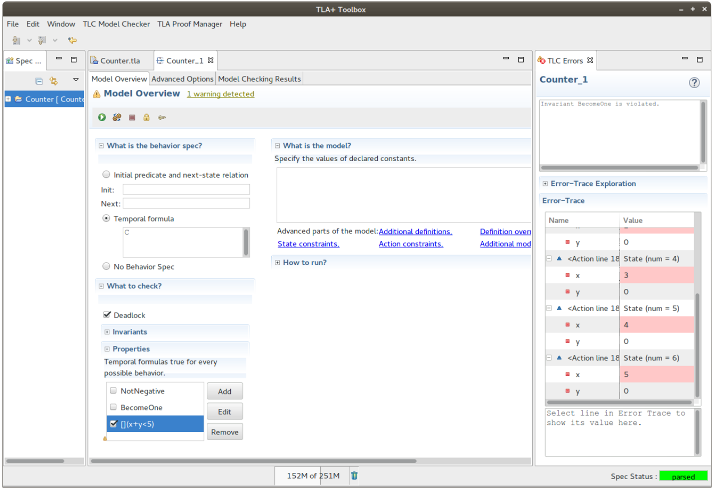
If necessary, restrict state space to finite subset.

The TLC Model Checker



Check the selected properties.

The TLC Model Checker



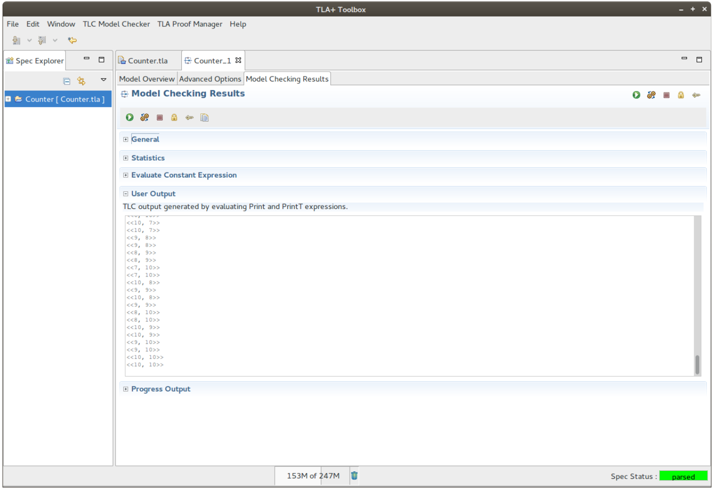
In the error case a violating system run is displayed.

Example

```
--------------- MODULE Counter ---------------
EXTENDS Naturals, TLC
VARIABLE x,y

...

C == I /\ [][R /\ PrintT(«x,y»)]_var /\ WF_var(X) /\ WF_var(Y)

...
=============================================================================
```

User output may help to validate the model.

The TLC Model Checker



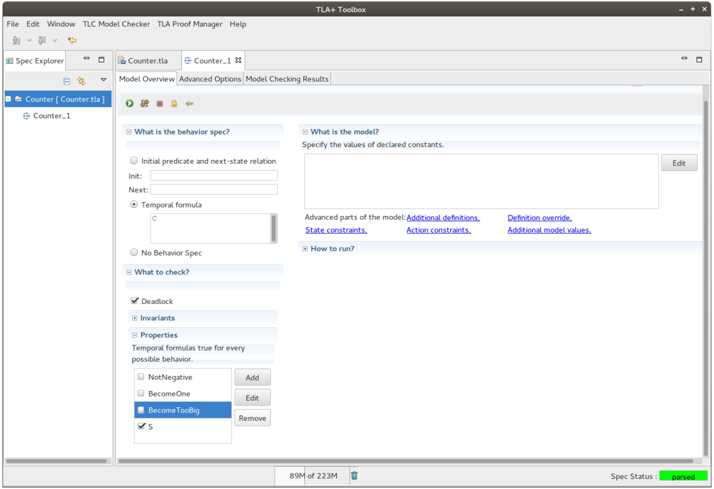
The visited states are printed.

The TLC Model Checker

```
--------------- MODULE Counter ---------------
EXTENDS Naturals
VARIABLE x,y

I == x = 0 /\ y = 0 (* the initial state condition *)

X == /\ x' = x+1 (* increment x *)
     /\ y' = y
Y == /\ x' = x   (* increment y *)
     /\ y' = y+1
R == \/ X        (* increment x or y *)
     \/ Y

var == «x,y» (* the system variables *)

C == I /\ [][R]_var /\ WF_var(X) /\ WF_var(Y)    (* the whole specification *)
S == (x = 0) /\ [][x' = x+1]_x /\ WF_x(x' = x+1) (* another system *)
=============================================================================
```

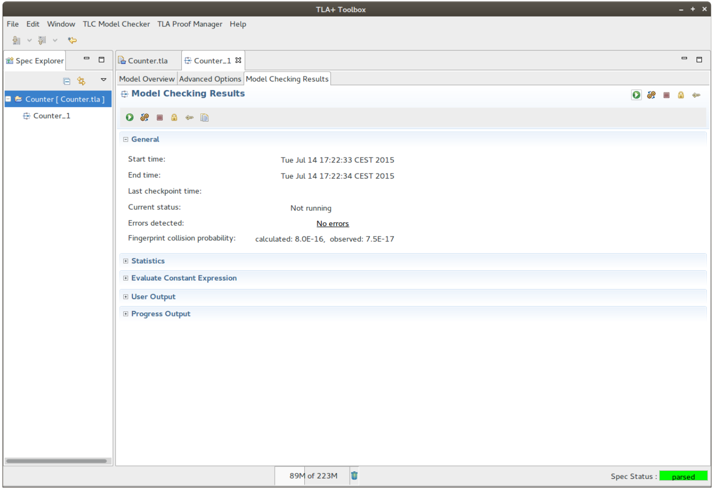
Specification of a more abstract system S.

The TLC Model Checker



Check whether C refines S ($C \Rightarrow S$).

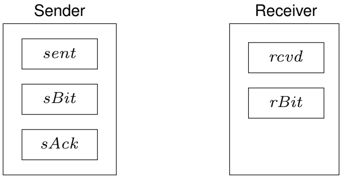
Der TLC Model Checker



System C is a valid refinement of S.

The Alternating Bit Protocol (Shared Memory)

Transmission of a sequence of bits between via shared registers.



var $sBit \in \{0,1\}, sAck \in \{0,1\}, rBit \in \{0,1\}, sent \in Data, rcvd \in Data$
init $sBit = sAck = rBit$

loop // Sender	||	**loop** // Receiver
wait $sAck = sBit$		**wait** $rBit \neq sBit$
$sent = \ldots; sBit = 1 - sBit$		$rcvd = sent; rBit = sBit$
		$sAck = rBit$

- Liveness property: $\forall d \in Data.$ *sent* $= d \wedge$ *sBit* $\neq$ *sAck* $\leadsto$ *rcvd* $= d$
  - Response: $P \leadsto Q \equiv \Box(P \Rightarrow \Diamond Q)$
  - Request P is always followed by response Q.

The Alternating Bit Protocol (Shared Memory)

MODULE *ABCorrectness*

EXTENDS *Naturals*
CONSTANTS *Data*
VARIABLES *sBit*, *sAck*, *rBit*, *sent*, *rcvd*

$ABCInit \triangleq sBit \in \{0, 1\} \land sAck = sBit \land rBit = sBit \land sent \in Data \land rcvd \in Data$

$$CSndNewValue(d) \triangleq \begin{array}{l} \land\ sAck = sBit \land sent' = d \land sBit' = 1 - sBit \\ \land\ \text{UNCHANGED}\ \langle sAck, rBit, rcvd \rangle \end{array}$$

$$CRcvMsg \triangleq \begin{array}{l} \land\ rBit \neq sBit \land rBit' = sBit \land rcvd' = sent \\ \land\ \text{UNCHANGED}\ \langle sBit, sAck, sent \rangle \end{array}$$

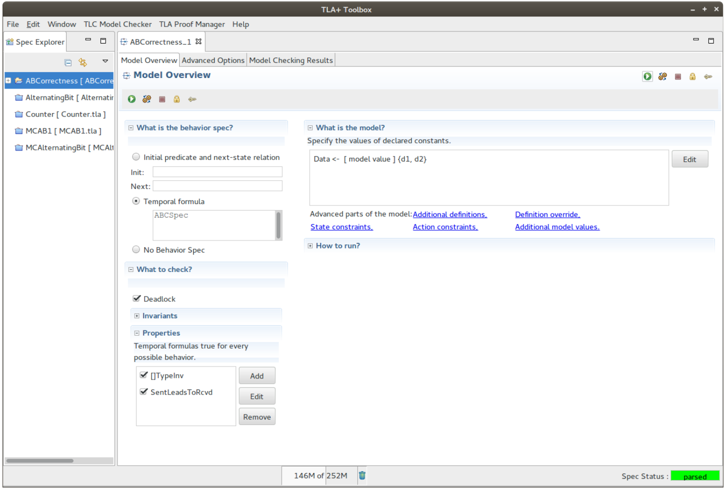
$$CRcvAck \triangleq \begin{array}{l} \land\ rBit \neq sAck \land sAck' = rBit \\ \land\ \text{UNCHANGED}\ \langle sBit, rBit, sent, rcvd \rangle \end{array}$$

$ABCNext \triangleq (\exists\, d \in Data : CSndNewValue(d)) \lor CRcvMsg \lor CRcvAck$

$cvars \triangleq \langle sBit, sAck, rBit, sent, rcvd \rangle$
$ABCSpec \triangleq ABCInit \land \Box[ABCNext]_{cvars} \land \mathrm{WF}_{cvars}(CRcvMsg) \land \mathrm{WF}_{cvars}(CRcvAck)$

$TypeInv \triangleq sBit \in \{0, 1\} \land sAck \in \{0, 1\} \land rBit \in \{0, 1\} \land sent \in Data \land rcvd \in Data$
$SentLeadsToRcvd \triangleq \forall\, d \in Data : (sent = d) \land (sBit \neq sAck) \leadsto (rcvd = d)$

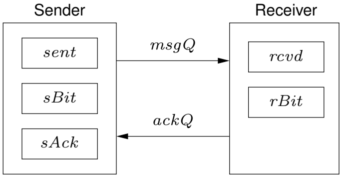
Model Checking the Protocol (Shared Memory)



No error: protocol satisfies specification.

The Alternating Bit Protocol (Distributed Memory)

Transmission of a sequence of bits by *lossy* communication channels.



- $msgQ$: transmits messages $\langle sBit, sent \rangle$.
  - New values after update by sender.
- $ackQ$: transmits messages $rBit$.
  - New values after update by receiver.

This protocol shall satisfy the same correctness property as the original one.

The Alternating Bit Protocol (Distributed Memory)

MODULE *AlternatingBit*

EXTENDS *Naturals*, *Sequences*

CONSTANTS *Data*

VARIABLES *msgQ*, *ackQ*, *sBit*, *sAck*, *rBit*, *sent*, *rcvd*

$$ABInit \triangleq \land\ msgQ = \langle\rangle \land ackQ = \langle\rangle$$
$$\land\ sBit \in \{0, 1\} \land sAck = sBit \land rBit = sBit \land sent \in Data \land rcvd \in Data$$

.....

$$ABNext \triangleq \lor\ (\exists\, d \in Data : SndNewValue(d))$$
$$\lor\ ReSndMsg \lor RcvMsg \lor SndAck \lor RcvAck \lor LoseMsg \lor LoseAck$$

$$abvars \triangleq \langle msgQ, ackQ, sBit, sAck, rBit, sent, rcvd\rangle$$
$$ABSpec \triangleq \land\ ABInit \land \Box[ABNext]_{abvars}$$
$$\land\ \mathrm{WF}_{abvars}(ReSndMsg) \land \mathrm{WF}_{abvars}(SndAck) \land \mathrm{SF}_{abvars}(RcvMsg) \land \mathrm{SF}_{abvars}(RcvAck)$$

$$ABTypeInv \triangleq \land\ msgQ \in Seq(\{0, 1\} \times Data) \land ackQ \in Seq(\{0, 1\})$$
$$\land\ sBit \in \{0, 1\} \land sAck \in \{0, 1\} \land rBit \in \{0, 1\} \land sent \in Data \land rcvd \in Data$$

INSTANCE *ABCorrectness*

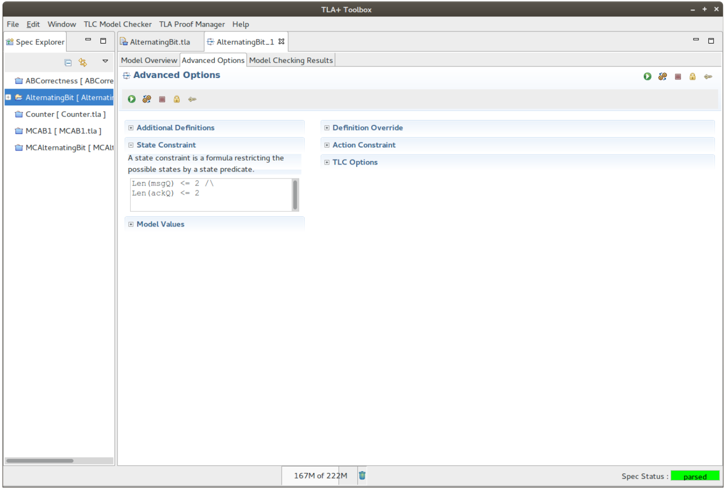
The core of the specification.

The Alternating Bit Protocol (Distributed Memory)

$$
\begin{array}{lcl}
SndNewValue(d) & \triangleq & \land\ sAck = sBit \land sent' = d \land sBit' = 1 - sBit \\
& & \land\ msgQ' = Append(msgQ, \langle sBit', d \rangle) \\
& & \land\ \text{UNCHANGED}\ \langle ackQ, sAck, rBit, rcvd \rangle \\
ReSndMsg & \triangleq & \land\ sAck \neq sBit \\
& & \land\ msgQ' = Append(msgQ, \langle sBit, sent \rangle) \\
& & \land\ \text{UNCHANGED}\ \langle ackQ, sBit, sAck, rBit, sent, rcvd \rangle \\
RcvMsg & \triangleq & \land\ msgQ \neq \langle\rangle \land msgQ' = Tail(msgQ) \land rBit' = Head(msgQ)[1] \land rcvd' = Head(msgQ)[2] \\
& & \land\ \text{UNCHANGED}\ \langle ackQ, sBit, sAck, sent \rangle \\
SndAck & \triangleq & \land\ ackQ' = Append(ackQ, rBit) \\
& & \land\ \text{UNCHANGED}\ \langle msgQ, sBit, sAck, rBit, sent, rcvd \rangle \\
RcvAck & \triangleq & \land\ ackQ \neq \langle\rangle \land ackQ' = Tail(ackQ) \land sAck' = Head(ackQ) \\
& & \land\ \text{UNCHANGED}\ \langle msgQ, sBit, rBit, sent, rcvd \rangle \\
Lose(q) & \triangleq & \land\ q \neq \langle\rangle \\
& & \land\ \exists\, i \in 1 \,..\, Len(q) : q' = [j \in 1 \,..\, (Len(q) - 1) \mapsto \text{IF}\ j < i\ \text{THEN}\ q[j]\ \text{ELSE}\ q[j+1]] \\
& & \land\ \text{UNCHANGED}\ \langle sBit, sAck, rBit, sent, rcvd \rangle \\
LoseMsg & \triangleq & Lose(msgQ) \land \text{UNCHANGED}\ ackQ \\
LoseAck & \triangleq & Lose(ackQ) \land \text{UNCHANGED}\ msgQ
\end{array}
$$

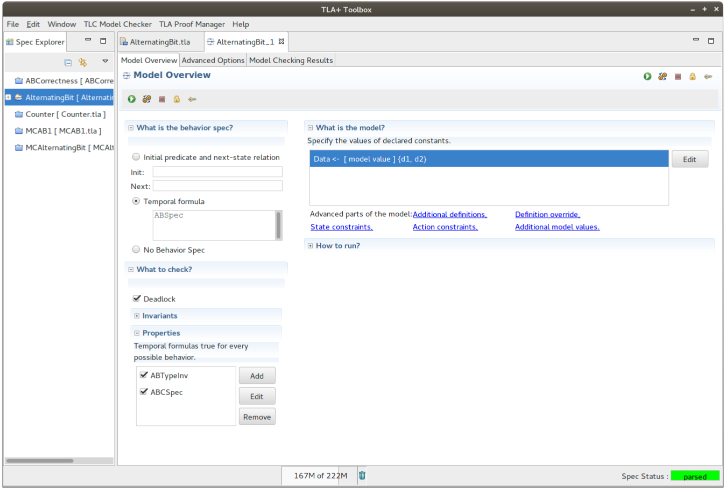
The actions of the specification.

State Space of the Protocol (Distributed Memory)



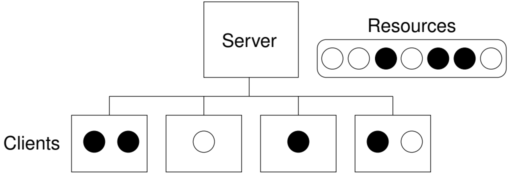
Restriction of the state space to a finite subset.

Model Checking the Protocol (Distributed Memory)



No error: the protocol refines the original one and thus inherits its correctness.

A Distributed Resource Allocator



- A server allocates various resources to a set of clients.
- A client with no resources and no pending requests may request some resources.
- The server may assign some or all of the requested resources.
  - Resource requests can be processed in multiple parts; the client may potentially continue already with some part.
- The client may return a subset of its resources; ultimately it must return all of them.
- Safety: no resource is simultaneously allocated to two clients.
- Liveness: each resource request is eventually satisfied.

A Distributed Resource Allocator

The method operates with the following variables.

- Server:
  - *unsat*$[c]$: the resources requested by client c but not yet allocated by the server.
  - *alloc*$[c]$: the resources requested by client c and allocated by the server.
  - *sched*: the list of clients with pending requests.
    - Older requests appear further ahead in the list and are preferably handled.
- Client c:
  - *requests*$[c]$: the resources requested by client c that it has not yet received.
  - *holding*$[c]$: the resources held by the client.
- Netzwerk:
  - *network* : the messages pending in the network.

Since messages may be still pending in the network, the server view may be different from the client view.

A Distributed Resource Allocator

MODULE *DistributedAllocator*

EXTENDS *Naturals*, *Sequences*
CONSTANTS *Clients*, *Resources*
VARIABLES *unsat*, *alloc*, *sched*, *requests*, *holding*, *network*

$$Messages \triangleq [type : \{\text{"request"}, \text{"allocate"}, \text{"return"}\}, clt : Clients, rsrc : \text{SUBSET } Resources]$$
$$Drop(seq, i) \triangleq SubSeq(seq, 1, i-1) \circ SubSeq(seq, i+1, Len(seq))$$
$$available \triangleq Resources \setminus (\text{UNION } \{alloc[c] : c \in Clients\})$$
$$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$$

$$\begin{aligned} Init \triangleq\ & \land unsat = [c \in Clients \mapsto \{\}] \land alloc = [c \in Clients \mapsto \{\}] \\ & \land requests = [c \in Clients \mapsto \{\}] \land holding = [c \in Clients \mapsto \{\}] \\ & \land sched = \langle\rangle \land network = \{\} \end{aligned}$$

$$\begin{aligned} Next \triangleq\ & \lor \exists m \in network : RReq(m) \lor RAlloc(m) \lor RRet(m) \\ & \lor \exists c \in Clients, S \in \text{SUBSET } Resources : Request(c, S) \lor Allocate(c, S) \lor Return(c, S) \end{aligned}$$

$$vars \triangleq \langle unsat, alloc, sched, requests, holding, network \rangle$$

$$\begin{aligned} Liveness \triangleq\ & \land \forall c \in Clients : \mathrm{WF}_{vars}(requests[c] = \{\} \land Return(c, holding[c])) \\ & \land \forall c \in Clients : \mathrm{WF}_{vars}(\exists S \in \text{SUBSET } Resources : Allocate(c, S)) \\ & \land \forall m \in Messages : \mathrm{WF}_{vars}(RReq(m)) \land \mathrm{WF}_{vars}(RAlloc(m)) \land \mathrm{WF}_{vars}(RRet(m)) \end{aligned}$$

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

The core of the specification.

A Distributed Resource Allocator

$RReq(m) \triangleq$
$\land\ m \in network \land m.type =$ "request"
$\land\ alloc[m.clt] = \{\}$ * don't handle request messages *prematurely*(!)
$\land\ unsat' = [unsat \text{ EXCEPT } ![m.clt] = m.rsrc]$
$\land\ network' = network \setminus \{m\}$
$\land\ sched' = \text{IF } m.clt \in Range(sched) \text{ THEN } sched \text{ ELSE } Append(sched, m.clt)$
$\land\ \text{UNCHANGED } \langle alloc, requests, holding \rangle$

$RAlloc(m) \triangleq$
$\land\ m \in network \land m.type =$ "allocate"
$\land\ holding' = [holding \text{ EXCEPT } ![m.clt] = @ \cup m.rsrc]$
$\land\ requests' = [requests \text{ EXCEPT } ![m.clt] = @ \setminus m.rsrc]$
$\land\ network' = network \setminus \{m\}$
$\land\ \text{UNCHANGED } \langle unsat, alloc, sched \rangle$

$RRet(m) \triangleq$
$\land\ m \in network \land m.type =$ "return"
$\land\ alloc' = [alloc \text{ EXCEPT } ![m.clt] = @ \setminus m.rsrc]$
$\land\ network' = network \setminus \{m\}$
$\land\ \text{UNCHANGED } \langle unsat, sched, requests, holding \rangle$

The receipt of messages.

A Distributed Resource Allocator

$$
\begin{array}{l}
Request(c, S) \triangleq \\
\quad \land\ requests[c] = \{\} \land holding[c] = \{\} \\
\quad \land\ S \neq \{\} \land requests' = [requests \text{ EXCEPT } ![c] = S] \\
\quad \land\ network' = network \cup \{[type \mapsto \text{“request”}, clt \mapsto c, rsrc \mapsto S]\} \\
\quad \land\ \text{UNCHANGED } \langle unsat, alloc, sched, holding \rangle
\end{array}
$$

$$
\begin{array}{l}
Allocate(c, S) \triangleq \\
\quad \land\ S \neq \{\} \land S \subseteq available \cap unsat[c] \\
\quad \land\ \exists\, i \in \text{DOMAIN } sched : \\
\qquad\quad \land\ sched[i] = c \\
\qquad\quad \land\ \forall\, j \in 1 \,..\, i-1 : unsat[sched[j]] \cap S = \{\} \\
\qquad\quad \land\ sched' = \text{IF } S = unsat[c] \text{ THEN } Drop(sched, i) \text{ ELSE } sched \\
\quad \land\ alloc' = [alloc \text{ EXCEPT } ![c] = @ \cup S] \\
\quad \land\ unsat' = [unsat \text{ EXCEPT } ![c] = @ \setminus S] \\
\quad \land\ network' = network \cup \{[type \mapsto \text{“allocate”}, clt \mapsto c, rsrc \mapsto S]\} \\
\quad \land\ \text{UNCHANGED } \langle requests, holding \rangle
\end{array}
$$

$$
\begin{array}{l}
Return(c, S) \triangleq \\
\quad \land\ S \neq \{\} \land S \subseteq holding[c] \\
\quad \land\ holding' = [holding \text{ EXCEPT } ![c] = @ \setminus S] \\
\quad \land\ network' = network \cup \{[type \mapsto \text{“return”}, clt \mapsto c, rsrc \mapsto S]\} \\
\quad \land\ \text{UNCHANGED } \langle unsat, alloc, sched, requests \rangle
\end{array}
$$

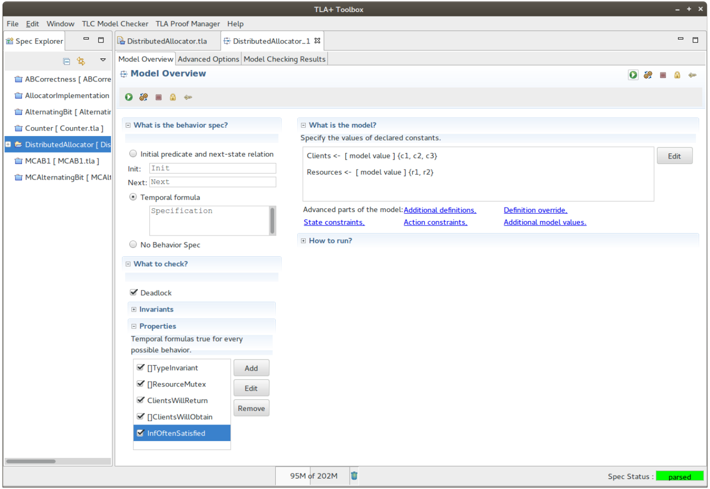
The sending of messages.

A Distributed Resource Allocator

$$
\begin{array}{l}
\mathit{TypeInvariant} \triangleq \\
\quad \land\ \mathit{unsat} \in [\mathit{Clients} \rightarrow \text{SUBSET}\ \mathit{Resources}] \land \mathit{alloc} \in [\mathit{Clients} \rightarrow \text{SUBSET}\ \mathit{Resources}] \\
\quad \land\ \mathit{requests} \in [\mathit{Clients} \rightarrow \text{SUBSET}\ \mathit{Resources}] \land \mathit{holding} \in [\mathit{Clients} \rightarrow \text{SUBSET}\ \mathit{Resources}] \\
\quad \land\ \mathit{sched} \in \mathit{Seq}(\mathit{Clients}) \land \mathit{network} \in \text{SUBSET}\ \mathit{Messages} \\
\mathit{ResourceMutex} \triangleq \\
\quad \forall\, c1, c2 \in \mathit{Clients} : \mathit{holding}[c1] \cap \mathit{holding}[c2] \neq \{\} \Rightarrow c1 = c2 \\
\mathit{ClientsWillReturn} \triangleq \\
\quad \forall\, c \in \mathit{Clients} : (\mathit{requests}[c] = \{\} \leadsto \mathit{holding}[c] = \{\}) \\
\mathit{ClientsWillObtain} \triangleq \\
\quad \forall\, c \in \mathit{Clients}, r \in \mathit{Resources} : r \in \mathit{requests}[c] \leadsto r \in \mathit{holding}[c] \\
\mathit{InfOftenSatisfied} \triangleq \\
\quad \forall\, c \in \mathit{Clients} : \Box\Diamond(\mathit{requests}[c] = \{\})
\end{array}
$$

The correctness properties.

Model Checking of the Distributed Resource Allocator



The allocator satisfies the correctness property (for 3 clients and 2 resources).